ANS MUMPS

Programmers' Reference Manual

MUMPS Users' Group 1981

Organizations Providing Generous Support for the Publication of this Book:

David A. Bridger, Inc.
6943 Waterman
St. Louis, MO 63130
(314) 862-1815

David A. Bridger, Inc. offers UNIX MUMPS, a MUMPS implementation that runs under UNIX Version 7 or an equivalent operating system; developed originally on the ONYX C8002, this implementation is available on a wide range of hardware (UNIX is a trademark of Bell Laboratories).

Compucare, Inc.
1970 Chain Bridge Road, Suite 602
McLean, VA 22102
(703) 821-8858

Compucare, Inc. specializes in the development and implementation of hospital information systems, serving over 75 hospitals across the country as system consultants and managers, and has developed several comprehensive on-line, real-time hospital systems using MUMPS.

Computer Resources, Inc.
6289 W. Sunrise Blvd., Suite 117
Ft. Lauderdale, FL 33313
(305) 791-6080

CRI offers Standard MUMPS and COSTAR on a full line of multilingual virtual memory systems supporting up to 128 terminals.

Innovative Computer Systems, Inc.
1660 S. Albion, Suite 902
Denver, CO 80222
(303) 759-3936

Innovative Computer Systems provides MUMPS for Data General computers, and provides business applications and contract work for all MUMPS systems.

Interactive Systems & Management Corp.
1500 Cardinal Drive
Little Falls, NJ 07424
(201) 256-7633

Interactive Systems & Management Corp. is a computer services company specializing in the development, installation, and support of MUMPS business systems; services include systems analysis and design, custom programming, time-sharing, turnkey system installation and support, and package software (including systems for pharmacy, mail-order, garment manufacturing, order entry, A/P, A/R, general ledger and text processing).

InterSystems Corporation
1 Commercial Wharf North
Boston, MA 02110
(617) 227-1555

InterSystems Corporation develops, maintains and markets Standard MUMPS implementations with optional distributed database capabilities (ISM) and a complete MUMPS-based word processing subsystem (MULTIWORD) that operate on a wide range of micro, mini, supermini and medium scale computers manufactured by Digital Equipment Corporation.

Medical Data Corporation
114 Airport Drive, Suite 105
San Bernadino, CA 92408
(714) 825-2683

Medical Data Corporation's MAXI-MUMPS™ is an ANSI conforming operating/integer system for Data General Eclipse computers, which support interactive on-line applications with a large number of concurrent terminals.

Micronetics Design Corporation
1145 19th Street, N.W., Suite 605
Washington, DC 20036
(202) 887-1948

Micronetics Design Corporation offers single-user and multi-user MUMPS systems for microprocessors (Motorola 6809 based systems and APPLE II systems), a complete line of hardware (CPUs, tapes, disks, printers, terminals, etc.) to support MUMPS applications as well as other applications, installation and support of the COSTAR medical information system, consulting services, and custom programming.

Plessey Peripheral Systems
17466 Daimler Avenue
Irvine, CA 92714
(714) 540-9945

Plessey Peripheral Systems manufactures, and markets worldwide, DEC based computer systems, including systems packaged with ISM-11, InterSystems's Standard MUMPS.

Plexus Computers, Inc.
2230 Martin Avenue
Santa Clara, CA 95050
(408) 988-1755

Plexus offers a range of high capacity 16-bit minicomputers running MUMPS under the UNIX operating system; multiple users are provided with high performance through dedicated I/O processors, high reliability through built in error checking and correction,and access to a wide variety of auxiliary devices through a Multibus I/O bus.

PURVIS Systems Incorporated
6901 Jerico Turnpike
Syosset, NY 11791
(516) 364-9690

PURVIS Systems, specialists in industrial laboratory information management systems, provides custom programming for interactive on-line applications, performs contract work and offers a full range of consulting services.

TANO Standard MUMPS
c/o Eclectic Systems Corporation
16260 Midway Road
Addison, TX 75001
(214) 661-1370

TANO Corporation and its subsidiary company Eclectic Systems Corporation have recently introduced a virtual memory implementation of ANSI Standard MUMPS, optimized to make the most efficient use of the mini floppy disk drives and Standard RAM available in TANO Corporation's Outpost series of microcomputers (OEM systems are available from TANO directly).

VISTA Computer, Inc.
709 Westchester Avenue
White Plains, NY 10604
(914) 428-3131

VISTA Computer, Inc. provides ANSI MUMPS on Data General equipment and DATASCAN, a general purpose data base management system.

ANS MUMPS
Programmers' Reference Manual 1981

and

ANS MUMPS Language Standard

Revision of the MUMPS Programmers' Reference Manual
by David Sherertz

Original Version by Melvin E. Conway

Edited by Ruth Dayhoff

Published by
MUMPS Users' Group
P.O. Box 37247
Washington, DC 20013

ACKNOWLEDGMENTS

The 1976 MUMPS Programmers' Reference Manual was written by Melvin E. Conway under Contract No. 5-35770 with the National Bureau of Standards. He was assisted by Paul L. Egerman who developed many of the programming examples, under NBS Purchase Order No. 512576. The work was authorized under the terms of an interagency agreement between the National Center for Health Services Research, Health Resources Administration, U.S. Department of Health, Education, and Welfare and the Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce.

The 1981 revision of the MUMPS Programmers' Reference Manual reflects extensions to the MUMPS Standard added as Type A releases by the MUMPS Development Committee since 1976. The revision was prepared by David Sherertz under contract with the MUMPS Users' Group and does not necessarily reflect the views of the MUMPS Development Committee which prepared the original text.

Because of the evolutionary nature of MUMPS specifications, the reader is reminded that changes are likely to occur in the specification reflected herein prior to a complete republication of MUMPS specifications.

NOTICE

Printed in U.S.A.

ISBN No. 0-918118-22-0

TABLE OF CONTENTS

* Each chapter indicated starts with a Table of Contents.

CHAPTER 1

PURPOSE OF THIS MANUAL

This document is an application programmers' reference manual for the use of the Standard MUMPS Language. Since it is written independently of any implementation of the MUMPS language, and since many implementations of MUMPS will be accompanied by their own application programmers' manuals, the question arises: What is the role of an implementation-independent reference manual? The significance of this question is heightened by the lack of a need for a teaching manual for the Standard MUMPS Language; the MUMPS Primer fills that role.

This reference manual addresses itself primarily to the programmer who already knows the Standard MUMPS Language well enough to write application programs. It serves such a programmer by means of the following approach.

1. The Reference Manual is for reference use. Once the use of the manual is understood (see Chapter 2), it is used to give specific answers to specific questions. After reading Chapter 2 and obtaining a familiarity with the rest of the document, the user should be able to dip into any portion of the manual and stay only until his specific question is answered.

2. The Reference Manual concentrates on questions of interpretation, rather than language structure. Syntax is presented, but the primary emphasis is on answering the "What happens when...?" type of question. As a result of this emphasis, much of the explanation of this manual is presented in terms of concrete actions of a "MUMPS System Model", described initially in Chapter 3. This MUMPS System Model is a hypothetical implementation of Standard MUMPS invented only for the expository purposes of this manual. Real implementations are not expected necessarily to imitate the internal characteristics of the System Model; however, if the MUMPS routines in question observe all MUMPS portability rules, a valid implementation of Standard MUMPS and the MUMPS System Model should give identical output, given identical input.

3. The Reference Manual attempts to illuminate the remote corners of the language as well as the most frequently used and most easily understood portions.

4. The Reference Manual encourages program portability by incorporating the MUMPS Portability Requirements. These requirements are not merely incorporated, however, but are explicitly cited. One important reason for visibly identifying the Portability Requirements is to avoid the appearance of contradicting the documentation of those implementations which relax some of these restrictions. By these means the responsibility for trading off portability against other benefits possibly offered by some implementations is placed where it belongs: with the management of the programming activity.

Secondarily, this Reference Manual addresses itself to the implementor. Implementors are constantly having to interpret language specifications by answering questions which were not anticipated by the language designers. This manual attempts to anticipate as many questions as possible. To cover the inevitable unanticipated questions, the use of a comprehensive model seeks to present a coherent structure within which different implementors will tend to make similar interpretations. This goal is particularly important to the designers of MUMPS, who have paid great attention to questions of application portability.

CHAPTER 2

A DESCRIPTION OF THE STYLE OF THIS MANUAL

Table of Contents

CHAPTER 2

A DESCRIPTION OF THE STYLE OF THIS MANUAL

2.1 Divisions of this Manual

As the Table of Contents indicates, this Reference Manual is divided into three parts.

Part I (Chapters 1-2) contains introductory material.

Part II (Chapters 3-6) presents a model of a MUMPS system. In Part II various concepts are introduced which are later employed in the language descriptions. Thus many of the cross-references in Part III refer back to Part II.

Part III (Chapters 7-12) describes the syntax and semantics of the elements of the MUMPS language, generally in top-down order, from routines (Chapter 7) to expression atoms (Chapter 12).

In addition, there are three appendices.

Appendix A contains a table of ASCII characters whose particular value to the MUMPS programmer is that the numeric character codes are presented in decimal form.

Appendices B and C contain indices to the definitions of the technical terms appearing in this manual.

2.2 Technical Terminology

Technical terms introduced in this manual fall mainly into two categories.

Syntactic Types

A syntactic type is a name given to a class of character strings which may appear in a MUMPS system. Membership in the class is defined by the spelling of the strings.

It is important to note that the strings classified by syntactic types can occur in several different contexts.

1. They may occur as part of a MUMPS routine. Examples are line, command, expression, string literal.

2. They may occur in data operated on by a MUMPS process. Examples are numeric value, nonnegative integer, truth value.

3. They may occur as values computed by a process defined in the model, but not necessarily appearing as data. Examples are node designator, device designator.

Thus, the syntax definition and specification methods employed in this manual apply uniformly to data strings as well as routine strings.

Syntactic types are always spelled entirely with lower-case letters (and an occasional hyphen). In certain contexts they may be underlined; this underlining does not change their nature, but is meant only to assist in rendering the accompanying description more readable.

1. When a syntactic type occurs in a syntax definition, it is always underlined. This underlining is particularly important when the type name has more than one word. Thus, expression atom is clearly a single entity.

2. In text, underlining a syntactic type is sometimes used to clarify a reference to a particular instance of that syntactic type. For example, in the syntax description of the SET command (8.2.15), an expression appears to the right of the = sign. Well down into the verbal description of the execution of the command the sentence appears: "The expression is evaluated". The use of underlining particularizes the reference so that it is clearly to the expression appearing in the syntax description.

Capitalized Technical Terms

The MUMPS System Model description introduces some technical terms, such as names of storage registers in the model, which are called here Capitalized Technical Terms. They are distinguished from syntactic types by appearing with initial capital letters in their spellings. Examples are Named System Storage, Job Number P-vector, Clock Register, Test Switch, Current Device Designator, Naked Indicator.

2.3 Syntax Specification Metalanguage

A syntax specification is a rule which partitions the set of all strings into two classes: those which satisfy the syntax specification and those which do not. Syntax specifications occur in two contexts.

1. On the right side of a syntax definition. A syntax definition has three parts: on the left is the name of the syntactic type being defined, in the middle is the metalangauge operator ::= , and on the right is a syntax specification. Examples from Chapter 7 follow.

line ::= line head line body

line head ::= [label] ls

line body ::= [command [␣ command] ... [␣ comment] | comment] eol

In the first example, the syntax type being defined is line, and the ::= says that, by definition, any (and only any) string which satisfies the syntax specification line head line body will satisfy the syntactic type line . Each syntactic type appears on the left side of at most one syntax definition (which may, however, be repeated within the manual). Those syntactic types which do not appear on the left side of a syntax definition, such as alpha, graphic, nonquote are defined informally; in the cases shown here (but not necessarily always) they are defined as a class of one-character strings.

2. As a free-standing syntax specification. For example, the definition of the syntactic type command is never formally given, but informally, command is defined to be satisfied by any of the syntax specifications of the separate commands described in 8.2. For example, the SET command has a (simplified) syntax specification as follows.

SET ␣ storage reference = expression

The following metalinguistic operations occur in syntax specifications.

Succession

Example: line head line body

Explanation: Any string which can be expressed as a concatenation of

1. a string satisfying line head on the left, and
2. a string satisfying line body on the right,

satisfies line head line body.

Choice

Example: | expression atom
| expression expression tail |

Explanation: The vertical lines are used to define the boundaries of a string under consideration, and have no other metalinguistic significance. Sometimes (as in the definition of line body given before) this function of boundary definition is performed by brackets [ ], which have an additional metalinguistic function as explained below.

Any string which satisfies either the syntax specification expression atom or the syntax specification expression expression tail satisfies the syntax specification

| expression atom
| expression expression tail | .

Option

Example: [label] ls

Explanation: Anything enclosed in brackets may be either present or absent. In the example, a string satisfies [label] ls either if it satisfies label ls or if it satisfies ls .

Repetition

Example: command | ␣ command | ... eol

Explanation: The three dots ... denote optional indefinite repetition of the immediately preceding minimum syntax specification, in this case ␣ command . Thus, a string satisfying any of the following satisfies the specification in the example.

command ␣ command eol

command ␣ command ␣ command eol

command ␣ command ␣ command ␣ command eol

etc.

Value Specification

Example: @ expression atom V global variable

Explanation: The metalinguistic operator V applies to the expression atom to its left and to the minimum syntax specification immediately to its right. It specifies that

1. the syntax of the syntax specification must satisfy @ expression atom , and
2. after the expression atom is evaluated, the syntax of its value must satisfy global variable .

Note: @ expression atom V is used to denote opportunities for indirection. Any discussion accompanying such a syntax specification applies to the string after all evaluation of indirection calls has occurred.

2.4 Starting Syntax Definitions

The following syntax definitions are taken as axiomatic.

graphic ::= the set of one-character strings whose characters are the 95 ASCII graphics

alpha ::= the set of one-character strings whose characters are the 52 ASCII alphabetics

_ ::= the one-character string consisting of the ASCII character space (SP)

Any ASCII graphic may appear in a syntax specification; it stands for the one-character string consisting of that character. (This provides another reason that syntactic types are always underlined when appearing in syntax specifications.) The following nonprinting ASCII characters may also occur in syntax specifications: space (represented by SP or ␣), carriage return (represented by CR), line feed (represented by LF), and form feed (represented by FF).

2.5 Portability Requirements

Statements taken from the MUMPS Portability Requirements are enclosed in the brackets

[Port: ... :Port] .

It is to be understood that such statements are not statements about the MUMPS language but are constraints which application programmers and MUMPS system implementors are asked to accept in order to make the goal of inter-system portability of application programs at reasonable cost a reality.

2.6 The Use of Examples

The text of this manual usually offers a precise basis from which the reader may make correct deductions. The examples, on the other hand, rely on the power of the reader to make correct inductions from statements which may not be formally complete or even correct. Thus the tone of the text accompanying the examples is more relaxed, and frequently there are inferences to be gained from the comparison of several examples.

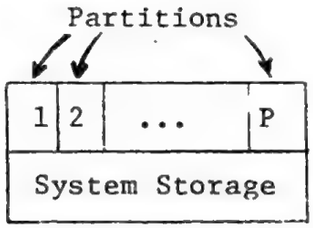
Ocassionally, the word "typically" appears in the text of this manual. Statements containing "typically" are not to be taken as precise but as describing common usage.

CHAPTER 3

THE MUMPS SYSTEM MODEL

For the positive integer P, a P-partition MUMPS system is one capable of supporting the concurrent execution of at most P MUMPS processes. Each "partition" is an environment for the execution of one process. The number P is usually a fixed (or very slowly changing) attribute of a given system, but this is not necessarily so.

The model employed here of a P-partition MUMPS system consists of P+1 components: the P partitions plus System Storage.



System Storage contains those data structures available to all partitions. The bulk of System Storage consists of what is traditionally called global data. In order to maintain a wide distinction between language entities and data, we shall use the term "Named System Storage" for that part of System Storage traditionally called global data.

In addition, System Storage contains a clock and three P-part data structures: the Lock List P-vector, the Open List P-vector, and the Job Number P-vector. Each partition has its own Lock List, Open List, and Job Number; the reason that they are assigned to System, and not Partition, Storage is that in manipulating the contents of its Lock and Open Lists each partition must know the contents of the P-1 other lists of the same type. Similarly, Job Numbers are assigned to be unique. The model is designed according to the premise that each partition may communicate with System Storage, but no two partitions may communicate directly with each other. The detailed structure of System Storage is described in Chapter 4.

Within each partition are the following components.

1. An interpreter
2. Partition Storage
3. The Partition Stack

Each partition is given its own interpreter in this model, not as an expression of a position on distributed vs. centralized processing, but as a way of avoiding having to take positions on time-slicing methods. As a further step in the direction of avoiding sequence dependencies arising from system (as opposed to application) design decisions, the interactions between the several interpreters and System Storage (reading values, writing values, killing subtrees, creating chains, creating and deleting lists, etc.) are defined to be indivisible and to occur nonsimultaneously. What is not defined here is how fast each interpreter works, either absolutely or in relation to any other interpreter.

Partition Storage contains those partition-local data structures which are not stacked by the DO or XECUTE commands; these include Named Partition Storage (i.e., local data or the "symbol table"), the Naked Indicator, the Test Switch ($T), and the Current Device Designator. Partition Storage is described in detail in Chapter 5.

The Partition Stack contains those entities which must be stacked for proper sequence control. In this model the whole routine body is stacked as well as status information such as the line pointer, line buffer, and character pointer. The model is not concerned with the details of stacking during expression evaluation, so there is no value stack as such in the model. Chapter 6 describes in detail the Partition Stack and its relationship to execution sequencing.

CHAPTER 4

SYSTEM STORAGE

Table of Contents

CHAPTER 4

SYSTEM STORAGE

System Storage in a P-partition system consists of the following elements.

1. The Named System Storage data structure.
2. The following P-vectors in each of which there is a one-to-one correspondence between the partitions and the components of the P-vector.
   a. The Job Number P-vector. Each element is an integer.
   b. The Lock List P-vector. Each element is a list of Named System Storage node designators or Named Partition Storage node designators.
   c. The Open List P-vector. Each element is a list of Device Designators.
3. The Clock Register.

4.1 Named System Storage

Named System Storage is a finite tree structure on every node of which resides an "attribute block" storage register whose contents are subject to certain constraints described below. The tree may be thought of as being arranged in levels wherein all the nodes at level n are at a distance n from the root ("directory") node.

Directory level (level 0)

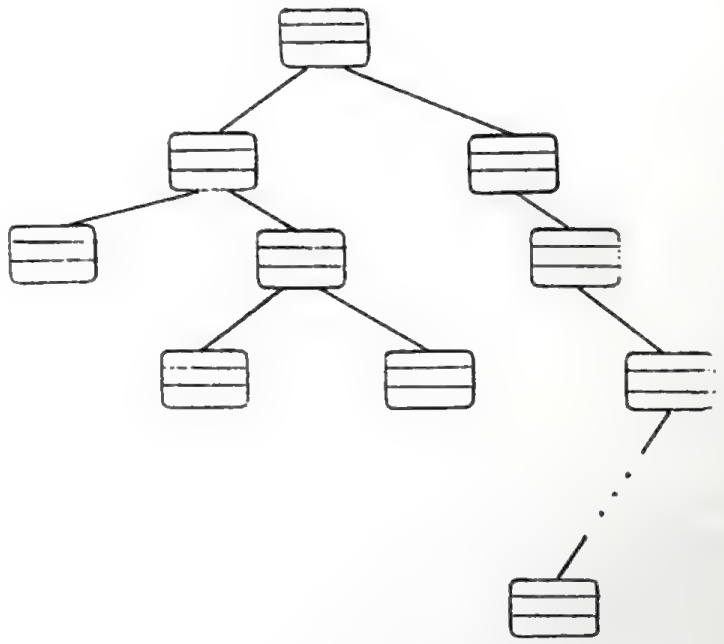
Unsubscripted level (level 1)

Singly subscripted level (level 2)

Doubly subscripted level (level 3)

.
.
.

n-tuply subscripted level (level n+1)



Each attribute block (shown as a rectangle above) is a three-part storage register containing the attributes named Name, D, Value.

Name:	^INV ¢ 3 ¢ 5
D:	1
Value:	2x4x8:160

Rules Governing Contents of Attribute Blocks At Each Node

1. The permissible entries in the D attribute are the four data strings
   0
   1
   10
   11
2. If D is 0 or 10, the Value attribute is said to be "undefined", and the content of the Value entry in the attribute block is unavailable. If D is 1 or 11, the Value attribute is said to be "defined", and the content of the Value entry in the attribute block may be any data string, including the empty string.
3. The Name attribute at level k (k≠0) is a k-tuple

   N ¢ S1 ¢ S2 ¢ ... ¢ Sk-1 , where

   a. The separator ¢ is chosen for typing convenience and denotes a unique separator outside of the data character set.
   b. Each element of the k-tuple is a data string.
   c. [Port: The spelling of each Si data string consists entirely of characters from the ASCII graphic subset (See 12.4). :Port] (Note: See 2.5 for a description of this portability notation.)
   d. The spelling of the N data string satisfies the syntax of ^name (see 12.4).

Note that the nodes or attribute blocks are considered to be part of the data structure, but the branches are not; the branches are drawn only to show ascendancy relationships (see below).

Rules Governing Coupling Between Attribute Block Contents and Tree Structure

1. Every node has an attribute block.
2. The attribute block on the node at the directory level (the "directory node") has only one entry of interest, D, whose contents may be only 0 or 10.
3. The initial state of the tree (prior to the first process initiation) is as follows: there is only the directory node, for which D=0.
4. In the following definitions, let N(i) denote any node at level i.
   a. If N(k) and N(k+1) are connected by a branch, then N(k+1) is called an "immediate descendant" of N(k) and N(k) is called the "immediate ascendant" of N(k+1).
   b. N(j) is an "ascendant" of N(k) and N(k) is a "descendant" of N(j) (for j < k) if and only if there is a sequence of nodes N(j), N(j+1), ..., N(k) such that for each i in [j,k-1] N(i) is the immediate ascendant of N(i+1).

   Rule: For any positive j, N(j) is an immediate ascendant of N(j+1) if and only if the content of the Name attribute of N(j) may be obtained from the content of the name attribute of N(j+1) by deleting ¢ Sj .
5. If the Node N(k) has D=0 or 1, then N(k) has no descendant. The converse is also necessarily true; N(k) may not be a "terminal node" (i.e., it has no descendant) with D=10 or 11.

4.2 Lock List P-vector

Each partition of the system, numbered 1 to P, is associated with a "Lock List". The collection of these P Lock Lists makes up the "Lock List P-vector" in System Storage. Each Lock List may be empty, or it may be a list of data strings satisfying the syntax of node designator and separated by a unique separator outside of the data character set, designated here by ¢¢.

node designator stands for the kinds of k-tuples (k≠0) which may occur in the Name part of the attribute block on a k-level node in either Named System Storage or Named Partition Storage:

N ¢ S1 ¢ S2 ¢ ... ¢ Sk-1 , where

a. The separator ¢ is outside of the ASCII data alphabet.
b. Each entry of the k-tuple is a data string.
c. [Port: The spelling of each Si data string consists entirely of characters from the ASCII graphic subset (see 12.4). :Port]
d. The spelling of the N data string satisfies the syntax of name or of ' name (see 12.4).

Lock List 1: [node designator [¢¢ node designator] ...]

.
.
.

Lock List P: [node designator [¢¢ node designator] ...]

Descendant Exclusivity Rule
No node designator in Lock List i may be an initial subsequence of any node designator in Lock List j, for any i≠j. (Interpretation: no node named in the Lock List of one partition may be an ascendant or descendant of any node named in the Lock List of any other partition.)

There is no necessary connection between the presence of any node designator in any Lock List and the existence of any node in Named System Storage or Named Partition Storage.

Initially, all Lock Lists are empty. In particular, when the process in partition i is initiated, Lock List i is empty. A Lock List is changed only as the direct result of one of the following actions.

1. Termination of the process (explicit or implicit execution of HALT) in partition i empties Lock List i.
2. Execution of LOCK without arguments in partition i empties Lock List i.
3. Execution of LOCK with an argument in partition i:
    a. first, empties Lock List i;
    b. then, sets the entire node designator list denoted by the argument into Lock List i, but only if the Descendant Exclusivity Rule will not thereby be violated. If the Rule would be violated, the Lock List remains empty.

4.3 Open List P-vector

The syntax of a "device designator" is specified by each system implementor; however, this much can be said here.

1. It is a data string.
2. It uniquely designates a "device", which is a sequential character source and/or a sequential character sink.

Each partition of the system, numbered 1 to P, is associated with an "Open List". The collection of these P Open Lists makes up the "Open List P-vector" in System Storage. Each Open List may be empty, or it may be a list of device designators separated by the nondata separator ¢¢.

Open List 1: [device designator [¢¢ device designator] ...]

.
.
.

Open List P: [device designator [¢¢ device designator] ...]

Device Exclusivity Rule
No device designator in Open List i may designate the same device as any device designator in Open List j, for any $i \neq j$. (Interpretation: partitions "own" devices exclusively.)

Initially, all Open Lists are empty. In particular, when the process in partition i is initiated, Open List i is empty. (See the discussion of the Principal Device Convention in 5.4 for an exception.) An Open List is changed only as the direct result of one of the following actions.

1. Execution of OPEN in partition i adds those device designators to Open List i which are denoted by the argument list.
2. Execution of CLOSE in partition i removes those device designators from Open List i which are denoted by the argument list. (See the discussion of the Principal Device Convention in 5.4 for consideration of a possible side effect of CLOSE.)
3. Termination of the process (explicit or implicit execution of HALT) in partition i empties Open List i.

4.4 Job Number P-vector

In order for a process to execute, it must be "active". The means by which a process becomes active is not defined in the Standard. Once a process is active, it remains active until it executes HALT or its equivalent (see 6.2 paragraph 2).

Each partition of the system which contains an active process is associated with a unique, positive-integer "Job Number". Inactive partitions have the Job Number zero. The collection of these P Job Numbers makes up the "Job Number P-vector" in System Storage. The Job Number of a process is assigned when the process becomes active, and it remains fixed throughout the life of the process. Its value is directly obtained by execution of the special variable $J as an expression atom. At no time may two active partitions have the same Job Number value. In some implementations the Job Number of an active partition may simply be the partition number, but this is not necessarily the case.

4.5 Clock

The content of this storage register is always defined and may be directly obtained by execution of the special variable $H as an expression atom. The clock value reflects the correct date and time and has the form of two integers separated by a comma: D,S . D is a day counter. S is a second counter running from 0 to 86399 for each value of D. D has a fixed value from each midnight to the immediately succeeding midnight; at the instant of midnight D is increased by 1 and S is set to 0. S counts seconds after midnight. The reference clock value 0,0 is the first second of January 1, 1841.

CHAPTER 5

PARTITION STORAGE

Table of Contents

CHAPTER 5

PARTITION STORAGE

In each of the P partitions of the system Partition Storage contains the following components.

1. The Named Partition Storage data structure, commonly called "local data" or the "symbol table".
2. The Naked Indicator.
3. The Test Switch.
4. The Current Device Designator.
5. The Current Device Horizontal Cursor and Current Device Vertical Cursor.

5.1 Named Partition Storage

In each partition, Named Partition Storage is a data structure which follows the identical set of rules as Named System Storage (see 4.1), except as follows.

Rule 3.d governing contents of attribute blocks at each node is changed as follows.

The spelling of the N part of the Name attribute, instead of having the syntax ^name , has the syntax name .

Rule 3 governing coupling between attribute block contents and tree structure reads as follows.

Each partition has its own distinct tree. Upon initiation of the process in partition i, its tree consists only of a directory node, for which D=0.

5.2 Naked Indicator

The content of the Naked Indicator is either empty ("undefined"), or it satisfies the syntax of node designator (see 4.2).

The Naked Indicator is changed only as a direct result of one of the following actions in the partition.

1. Initiation of the process causes the Naked Indicator to be undefined.
2. Execution of an unsubscripted global variable causes the Naked Indicator to be undefined.
3. Except for the above, execution of a global variable or naked reference causes the content of the Name attribute of the implied immediate ascendant to be copied into the Naked Indicator. That is, the remainder after removing the last ç Sk-1 from the implied level k node designator is placed into the Naked Indicator.

Execution of a naked reference is erroneous when the Naked Indicator is undefined. If the Naked Indicator is defined, execution of the naked reference ^(expression 1,...,expression i), where S1 = the value of expression 1, ..., Si = the value of expression i, implies the node designator

content of Naked Indicator ¢ S1 ¢ ... ¢ Si ,

after the use of which the Naked Indicator is given the value obtained by deleting ¢ Si.

5.3 Test Switch

The content of the Test Switch register is one of three data strings: empty ("undefined"), 0 ("false"), or 1 ("true"). When defined, this content may be directly obtained by execution of the special variable $T as an expression atom.

The content of the Test Switch is changed only as a direct result of one of the following actions in the partition.

1. Execution of IF with an argument places the truth-value interpretation of the value of the argument into the Test Switch.
2. Execution of LOCK, OPEN, or READ with an argument which has a timeout attached to the argument implies a test of some condition; specifically, the ability to load the partition's Lock List without violating the Descendant Exclusivity Rule, the ability to load the partition's Open List without violating the Device Exclusivity Rule, or explicit termination of the input data string prior to resumption of execution after the timeout, respectively. If the condition is satisfied, the Test Switch receives the value 1; otherwise it receives the value 0.

In addition to the ability to obtain directly the content of the Test Switch by execution of $T, the process may, by execution of ELSE or IF, conditionalize execution of the remainder of the line containing the ELSE or IF depending on the value of the Test Switch.

5.4 Current Device Designator

See the first paragraph of 4.3 for a definition of "device" and "device designator". The content of the Current Device Designator storage register is either empty ("undefined") or else it is a data string which satisfies the syntax of device designator. The content of the Current Device Designator may be directly obtained by executing the special variable $I as an expression atom.

The "current device" is that device which is uniquely designated within the partition for data transfer upon execution of READ or WRITE. When the designation of the current device is undefined (i.e., the content of the Current Device Designator is empty), the result of execution of READ or WRITE is undefined. Otherwise, the Current Device Designator designates the current device.

Current Ownership Rule
At all times the current device of partition i must also be designated in Open List i. (Interpretation: the process must own a device before it may be declared current. Also, when the process CLOSEs a device which is current, the device designator is removed from the Current Device Designator as well as from the Open List.)

The implementor has an option with respect to the initial content of the Current Device Designator and the response to CLOSE naming the current device. Following is a description of the "Principal Device Convention", which is one choice.

1. There is a device D for this process called the "Principal Device". The identity of the Principal Device is specified by the implementor and it does not change throughout the life of the process.
2. Upon initiation of the process in partition i, Open List i designates D (and D only), and partition i's Current Device Designator designates D.
3. Whenever a CLOSE is executed which would empty the Current Device Designator, a designator of D is placed into the Current Device Designator and the Open List (if it is not already there and if the operation can be performed without violating the Current Ownership Rule).

If the implementor does not follow the Principal Device Convention with respect to this process, the Current Device Designator will be empty initially and as a result of executing a CLOSE which names the current device.

Except as specified above, the content of the Current Device Designator is changed only as a direct result of execution of USE; the device designator specified by the argument of the executed USE replaces the content of the Current Device Designator, provided that the specified device designator designates a device also designated by the Open List.

5.5 Current Device Horizontal and Vertical Cursors

As part of the status information stored for each device in System Storage are two registers per device, X and Y, called the horizontal cursor and vertical cursor, respectively. In Partition Storage there are two registers, CX and CY, called the Current Device Horizontal Cursor and the Current Device Vertical Cursor, respectively, which make available the values of X and Y for the current device. The contents of CX and CY are directly available by execution of the special variables $X and $Y, respectively, as expression atoms.

When the content of the Current Device Designator register changes, the contents of CX and CY are placed into the X and Y registers of the former current device and the X and Y registers of the new current device are placed into the CX and CY registers. Thus, as ownership of a device changes, the state of its cursors is not lost.

The content of each cursor is a data string which satisfies the syntax of integer literal. The initial value of each cursor is not defined until the # format is executed by some process while the device is current in that process.

Aside from the changing of CX and CY arising from changing the current device, CX and CY are changed only by format operations and character transfers performed during the execution of READ and WRITE. Each format operation affects the values of CX and CY as follows.

!	places 0 into CX and adds 1 to CY.
#	sets CX and CY to 0.
?n	places the value of max(n,CX) into CX.

The effects on CX and CY resulting from data transfers performed during the execution of READ and WRITE are specified in the MUMPS Language Standard as goals; the framers of the Standard recognized that some implementations may exist within the constraints of operating systems which may make absolute achievement of all the goals impractical. The goals are stated here, and each implementor is expected to come as close to 100% achievement as is practical.

1. The effect on CX and CY of transferring a character is registered when that character is transferred.
2. The effect on CX and CY of transferring a character is a function of each character's identity, and is independent of the context of the character or whether READ or WRITE is being executed.
3. The following effects on CX and CY are specified.

Each graphic:	Add 1 to CX.
Backspace (BS):	Set CX = max(0,CX-1).
Line Feed (LF):	Add 1 to CY.
Carriage Return (CR):	Set CX = 0.
Form Feed (FF):	Set CX = 0, CY = 0.

CHAPTER 6

THE PARTITION STACK AND CONTROL OF EXECUTION SEQUENCE

Table of Contents

CHAPTER 6

THE PARTITION STACK AND CONTROL OF EXECUTION SEQUENCE

The Partition Stack is a last-in, first-out data structure which controls the sequence of execution of a MUMPS process. Each item of the stack contains the following five components.

1. A routine body.
2. A Line Pointer which designates one line of the routine body. It can also designate the eor at the end of that routine.
3. A Line Buffer which contains a (possibly modified) copy of the line designated by the Line Pointer.
4. A Character Pointer which designates one character of the Line Buffer.
5. A two-valued indicator called the For Scope switch.

[Port: Routines designed for portability should not cause the Partition Stack to be nested beyond fifteen levels. :Port]

6.1 Normal Execution Sequence

The execution of a MUMPS process is a sequence of atomic operations, each such atomic operation being the execution of one character. The routine body whose characters are currently being executed is always that at the top of the stack. Within this routine body the sequence of characters being executed is described by rules given in the remainder of this chapter. Normally, the following rules apply.

1. The "normal execution sequence" prevails except when it is altered by execution of HALT, IF, ELSE, FOR, QUIT, eor, by execution of an argument of DO, GOTO, IF, or XECUTE, or by indirection. Rules 2 and 3 describe the normal execution sequence.
2. Execution of a given line is defined to be the sequence of character executions which occur while a copy of that line is in the Line Buffer. When the Line Pointer is changed to designate a new line, the following occurs.
    a. A copy of the new line, from the character immediately to the right of ls to and including the eol, is taken from the routine body and placed into the Line Buffer.
    b. The Character Pointer is set to designate the leftmost character of the Line Buffer.
3. A copy of the character designated by the Character Pointer is obtained and executed. If the character is eol and the For Scope switch is off, its effect is to cause the Line Pointer to advance to the next line (or to the eor). If the character is not eol, then just prior to execution of the character, the Character Pointer is advanced to the next character to the right; then the character which had been obtained is executed.

The normal execution sequence, then, is a strict left-to-right sequence of execution of characters within lines and of lines within the routine body. The identity of the first line to be executed when a new routine body is pushed onto the top of the Partition Stack is specified within the DO argument which names the routine. In the default case, i.e., when the DO argument contains no line specification or upon initiation of the process without a line specification, the first line of the routine body is implied.

In the initial state of a process just prior to execution of the first character, the Partition Stack contains one routine body, the Line Pointer and the Character Pointer designate the first character of the first executed line of the routine body, and the For Scope switch is off.

6.2 Changes of Sequence

Deviation from the normal execution sequence can occur in the following ways.

1. Execution of HALT causes the process to terminate. Termination of a process implies the following operations in System Storage associated with the partition k in which the process terminates.
    a. The kth element of the Job Number P-vector is set to zero.
    b. The kth element of the Lock List P-vector is made empty.
    c. The kth element of the Open List P-vector is made empty.

2. Execution of QUIT with the For Scope switch off, or execution of eor, causes the top element of the Partition Stack to be popped. If the Partition Stack thereby becomes empty, the process terminates as if HALT had been executed. If there is still a routine body on the Partition Stack, the next character executed is the character designated by the Character Pointer in the new stack top.

3. Execution of any of the following subject to the condition stated causes the Character Pointer to scan to the right fetching but not executing each character until it points to the eol of the Line Buffer; that is, the effect of executing any of the following is to suppress execution of the remainder of the line to its right.
    a. Execution of argumentless IF such that the content of the Test Switch is 0.
    b. Execution of ELSE such that the content of the Test Switch is 1.
    c. Execution of an IF argument whose value (after taking the truth-value interpretation) is zero.

4. Execution of a DO argument causes a new element to be pushed onto the Partition Stack. The procedure is described as follows.
    a. If the DO argument contains a routine name.
        (1) The routine body is that designated by the routine name.
        (2) The Line Pointer is set to point to the line designated by the DO argument; if the DO argument contains no line reference, the Line Pointer is set to point to the first line of the routine body.
        (3) The Line Buffer is loaded and the Character Pointer is set to point to its leftmost character.
        (4) The For Scope switch is set off.
    b. If the DO argument contains no routine name.
        (1) The routine body is a copy of the one at the top of the stack.
        (2) The Line Pointer is set to point to the line designated by the DO argument. Then (3) and (4) above are executed.

5. Execution of a GOTO argument causes the following two operations to occur in sequence.
    a. The top element of the stack is popped as if QUIT with the For Scope switch off had been executed.
    b. An element is pushed onto the stack as if a DO argument with the same spelling as the GOTO argument had been executed.

6. Execution of an XECUTE argument is similar to execution of a DO argument with the following amendments.
    a. The routine body is a copy of the one at the top of the stack.
    b. The Line Pointer is set to point to the last line of this routine body, just preceding the eor.
    c. The Line Buffer is loaded with the following string

        value of XECUTE argument eol

        and the Character Pointer is set to its leftmost character.

7. Execution of the command word FOR causes the For Scope switch to be turned on, regardless of its prior state. After evaluation of all expressions in the first for parameter, a unique marker character, called a Parameter Marker, is inserted into the Line Buffer between the for parameter just evaluated and the separator immediately to its right. The Character Pointer is then moved to the right to the first character to the right of the space following the for parameter list. (If the for parameter list is immediately followed by eol, the Character Pointer points to this eol.) Another unique character, called a Scope Marker, is inserted immediately to the left of this character, and normal execution resumes. The following subsequent breaks in the normal sequence can occur.
    a. If eol is executed with the For Scope switch on and the scope must be executed again under the current for parameter, the Character Pointer repositions at the character to the right of the rightmost Scope Marker, and the normal execution sequence resumes.
    b. If eol is executed with the For Scope switch on and the scope does not have to be executed again under the current for parameter, the Character Pointer scans to the left looking for a Parameter Marker.
        (1) If a Parameter Marker is found and the character immediately to its right is a space or eol (that is, the scope of the FOR under control of the last for parameter has just been executed), then the Parameter Marker and the Scope Marker to its right are deleted and the leftward scan described in b. above is resumed.
        (2) If no Parameter Marker is found, the For Scope switch is turned off and the normal eol execution described in 6.1 paragraph 3. above occurs.
        (3) If a Parameter Marker is found and the character immediately to its right is a comma, the Parameter Marker is deleted, the for parameter to its right is evaluated, the Parameter Marker is inserted immediately to the right of this for parameter, the Character Pointer is moved to point to the character immediately to the right of the Scope Marker, and normal execution resumes.
    c. Execution of QUIT with the For Scope switch on causes the Character Pointer to scan to the left until it encounters a Parameter Marker. This Parameter Marker and the Scope Marker to its right are deleted. Then the eol procedure beginning at 7a. above is executed.
    d. Execution of a GOTO argument with the For Scope switch on is unchanged from that described in 5.

8. Execution of @ (signaling indirection) causes the following steps to occur.
    a. The Character Pointer continues to the right as the expression atom to the right of the @ is evaluated. Note that this evaluation can encounter a @, so the whole of 8. is recursive.
    b. After the expression atom is evaluated, the value of the expression atom is inserted into the Line Buffer immediately to the right of the expression atom, and the Character Pointer is set to point to the leftmost character of this value. (If the value is empty, the Character Pointer points to the first character to the right of the expression atom, and the effect of the following step is null.)
    c. As each character in the Line Buffer which arose from the evaluation of the expression atom is designated by the Character Pointer and a copy of it is fetched, that character is deleted from the Line Buffer. The effect is to leave no trace that indirection has occurred so that a single instance of indirection may be called repeatedly in the scope of a FOR.

CHAPTER 7

ROUTINES AND LINES

The routine is the unit of algorithm interchange. Programs and packages are unordered collections of routines. When called for execution, a routine is denoted by its routine name. The syntax observed by routine names is the same as name.

routine name ::= | % alpha | [digit alpha] ...

[Port: Routine names containing 9 or more characters are identified and distinguished only on the basis of their left 8 characters. No lower-case alphabetic characters may occur in a routine name. :Port]

When one refers to the syntax of a routine, one is referring to the routine as stored on a linear external medium as in algorithm interchange. In this context, a routine consists of two parts: the head, which contains the routine name for identification in interchange, and the body, which contains the executed code, consisting of a sequence of lines.

routine ::= routine head routine body

routine head ::= routine name eol

routine body ::= line [line] ... eor

In interchange, the three separators ls, eol, and eor denote the following ASCII character or control character pairs.

ls ::= SP

eol ::= CR LF

eor ::= CR FF

Thus, if the routine is printed on a device which observes ASCII control conventions, the routine name would appear as the first printed line and the executed code would appear on the remaining printed lines.

Analogously to the structure of the routine, each line consists of two parts. The line head may contain an optional label for identifying the line, and the line body contains the executable commands, if any, of the line.

```
line       ::= line head  line body

line head  ::= [label]  ls

                ⎡ command  [ ␣ command ] ... [ ␣ comment ] ⎤
line body  ::=  ⎢                                          ⎥  eol
                ⎣              comment                     ⎦

               | name           |
label      ::= |                |
               | integer literal |

comment    ::= ; [graphic] ...
```

[Port: The total number of characters in a line, excluding the eol, but including the remainder of the line body and the whole of the line head, may not exceed 255. :Port]

Labels are used to identify lines. Line references occur in DO and GOTO commands and in the $TEXT function. DO and GOTO may refer to a line in any routine, whereas $TEXT refers only to the routine body at the top of the Partition Stack.

Within a given routine, a line whose line head contains a label may be designated by naming the label. Thus the simplest form of a line reference is an occurrence of label. More generally, any line in a given routine may be designated by referring to a prior line containing a label in its line head. The form label + integer expression (where i is the integer interpretation of the value of the expression) means "the ith line after the line containing label in its line head". The reference label + 0 is equivalent to label. Negative values of integer expression are not allowed.

Within any given routine, no spelling of any label may occur more than once in any line head. When the label consists entirely of numeric digits, leading zeros are significant to its spelling; that is, the label 01 is different from the label 1. [Port: Labels containing 9 or more characters are identified and distinguished only on the basis of their left 8 characters. :Port]

In the general form of a line reference, indirection is permitted on the label portion but not on the line reference as a whole.

line reference ::= dlabel [+ integer expression]

dlabel ::= | label | @ expression atom V dlabel |

The $TEXT function has a default definition which handles the case of a line reference either reaching beyond the end of a routine or referring to a label not appearing in a line head in the routine. DO and GOTO do not have such default definitions; therefore the following conditions are not allowed in the context of DO or GOTO.

1. A spelling of label which does not occur in a line head of the routine denoted.
2. A value of integer expression so large as not to denote a line within the routine body.

DO and GOTO can specify a routine as well as a line, and their arguments therefore contain a more general form of line reference called an entry reference. Within an entry reference, three syntactic cases may be distinguished.

1. The entry reference has the form of line reference , in which case the routine body implied is that at the top of the routine stack.
2. The entry reference has the form of ^routine reference , in which case the first line of the routine whose name is denoted by the routine reference is implied.
3. The entry reference has the form of line reference^routine reference , in which case the designated line of the designated routine is implied.

entry reference ::= | line reference [^ routine reference] | ^ routine reference |

routine reference ::= | routine name | @ expression atom V routine reference |

CHAPTER 8

COMMANDS

Table of Contents

CHAPTER 8

COMMANDS

8.1 Introductory Definitions

8.1.1 Command Syntax

In Chapter 7 it is shown how the syntactic object command occurs within a line body. In this chapter the syntax and semantics of command are made explicit.

In place of any occurrence of command in the definition of line body can stand any of the command definitions of 8.2. Every command begins with a mnemonic command word spelled entirely with capital letters, such as SET.

Commands fall into two basic categories. Some command words have variations which fall into both categories.

1. Argumentless commands. The syntax of an argumentless command consists of the command word, possibly followed by a space.
   a. If the occurrence of an argumentless command immediately precedes the eol, there is no space following the command word.

      QUIT eol

   b. If the occurrence of an argumentless command does not immediately precede the eol, the command word is followed by exactly two spaces, the first considered to be part of the command, and the second considered to be between this command and the next command or comment. (Refer to the syntax of line body in Chapter 7.)

      ELSE ␣␣ GOTO ␣ X eol

2. Commands with arguments. The syntax of a command which takes an argument consists of the command word followed by a single space followed by an argument or argument list.

   IF ␣ X=Y ␣ DO ␣ A ␣ QUIT:X=1 ␣␣ DO ␣ B eol

8.1.2 Format of Command Definitions

Each command definition of 8.2 is organized into the following parts.

1. Syntax. The syntax of the most suggestive, and usually the simplest, form of the command is given here.
2. Elaboration. Variations on the syntax given in 1. are discussed. These can fall into the following five categories discussed in detail in 8.1.3.
    a. Abbreviation of the command word
    b. Post-conditionalization of the command word
    c. Listing of multiple arguments
    d. Post-conditionalization of any argument
    e. Argument indirection
3. Execution. The effect of executing the command is described.
4. Cross-reference. Reference is made to other portions of this reference manual where a related discussion may shed light on the definition of this command.
5. Examples. Examples of use of the command are given, usually with explanation and in a sequence which permits the reader to make useful inferences about the command.

8.1.3 Elaborations

8.1.3.1 Abbreviation of the Command Word

Wherever a command word may properly occur, it may occur in either of two forms: as fully spelled out in the syntax portion of the command definition, or in an initial-letter abbreviation. Other forms are not allowed. For example,

GOTO ␣ A

may be written

GOTO ␣ A

or

G ␣ A

but neither

GO ␣ A

nor

GOT ␣ A

nor

GOING ␣ A

nor

GOTOHERE ␣ A .

8.1.3.2 Post-conditionalization of the Command Word

A post-conditional is a colon immediately followed by a truth-valued expression, the pair of which can appear as an optional suffix to certain command words or command arguments.

post-conditional ::= : truth-value expression

All command words defined by the MUMPS Standard except ELSE, FOR, and IF may be post-conditionalized, either in their spelled-out or abbreviated forms. For example, a post-conditionalized form of

```
SET ␣ X=1
```

is

```
SET:Y=2 ␣ X=1
```

or

```
S:Y=2 ␣ X=1
```

.

(Note the two different uses of the "=".)

The interpretation of the post-conditional attached to a command word is as follows.

1. If the post-conditional is absent, the command as a whole is executed unconditionally, subject to external factors which might inhibit its execution, such as a preceding ELSE or IF.
2. If the post-conditional is present, it is evaluated before anything to its right is executed. If the truth-value interpretation of its value is 1 (true), then the command as a whole is executed. If the truth-value interpretation of the value of the post-conditional is 0 (false), then no part of the command is executed and no argument is evaluated.

The evaluation of a post-conditional can have side effects even if it inhibits the execution of its host command. For example,

```
SET:$D(^(1,2))=3 ␣ X=X+1
```

will never change X but will always change the Naked Indicator.

Since the post-conditional does not affect the Test Switch, its use is not equivalent to the use of IF.

8.1.3.3 Listing of Multiple Arguments

The syntax presented in 8.2 for each command which accepts an argument has the following structure.

command word a single space a single argument

If the elaboration does not state explicitly that arguments may be listed, then the argument form shown describes all permissible argument structures for this command. If the elaboration states that arguments may be listed, then the following convention applies.

Let Y denote the command word, and let A1, A2, ..., An each denote a single argument, each with syntax as specified in the syntax description. If the syntax description is shown as

Y ␣ A1

then the following forms of the command syntax are also permitted.

Y ␣ A1,A2

Y ␣ A1,A2, ... ,An .

(The three dots are not to be taken literally but suggest indefinite but finite repetition, with a comma and no space between each pair of arguments.)

The interpretation of

Y ␣ A1,A2, ... ,An

is precisely the same as

Y ␣ A1 ␣ Y ␣ A2 ␣ ... ␣ Y ␣ An .

That is, each comma separating command arguments preceded by the command word Y is literally to be interpreted as

␣ Y ␣ .

[Port: The restriction on the number of characters in a line applies as the line actually appears in routine interchange, and not as the line might appear after making each substitution of ␣ Y ␣ for , :Port]

8.1.3.4 Post-conditionalization of Any Argument

Only the commands DO, GOTO, and XECUTE permit post-conditionalization of arguments; therefore the following discussion applies only to these three commands.

Any argument may be post-conditionalized, independently of the post-conditionalization of the command word or of any other argument in the argument list. The interpretation of a post-conditionalized argument is as follows.

1. In what follows, the command word may be assumed to be not post-conditionalized, for this reason. If a post-conditional on the command word is present and (its value is) false, then no argument interpretation is necessary, since no argument of the command is interpreted or executed. If a post-conditional on the command word is present and true, then each argument is interpreted in turn as if the command word had not been post-conditionalized.
2. Let Pi denote a post-conditional appearing on argument Ai. With respect to the question of whether or not the explicit execution of a particular argument occurs,

   Y ␣ A1:P1

   is interpreted as

   Y:P1 ␣ A1 .

   In the interpretation of the expansion of

   Y ␣ A1,A2, ... ,An

   if any argument is of the form

   Ai:Pi ,

   then in the expansion in which the commas are removed it appears as

   Y:Pi ␣ Ai .

3. With respect to side effects, the matter is complicated by the fact that a post-conditional has no effect on the normal execution sequence (see Chapter 6) until after it, and the argument which it qualifies, have already been evaluated. Therefore, in the execution of

```
DO ␣ X:A
```

the entry reference X and the expression A are always fully evaluated (unless the DO is falsely post-conditionalized) even if A is false and therefore the subroutine call is not performed. Therefore,

```
XECUTE ␣ ^(3,4):0
```

will have the effect of attempting to change the Naked Indicator and of attempting to fetch the designated value (each of which attempts can lead to an error if the respective arguments are not defined), but not of executing the evaluated expression.

8.1.3.5 Argument Indirection

The following discussion applies to those commands for which argument indirection is permitted. Any command admitting argument indirection also admits argument listing.

The indirection call

```
@ expression atom
```

may occur in place of any argument in an argument list, or in place of any isolated argument. Thus, where the syntax permits

```
Y ␣ A1
```

then

```
Y ␣ @ expression atom 1
```

is permitted, and where the syntax permits

```
Y ␣ A1,A2, ... , An
```

then

```
Y ␣ A1,A2, ... ,@ expression atom i, ... ,An
```

is permitted.

The value of expression atom i must be an argument or argument list satisfying the syntax specification of Y. Thus, the value of expression atom 1 in Y _ @ expression atom 1 may be a single argument

A1

or an argument list

A1,A2, ... , An

or an argument list containing one or more indirection calls

A1,A2, ... ,@ expression atom i, ... ,An

requiring subsequent indirection evaluation. These three possibilities are equally valid for an @ expression atom i occurring anywhere in an argument list, since the expansion of

Y ␣ A1,A2, ... ,@ expression atom i, ... ,An

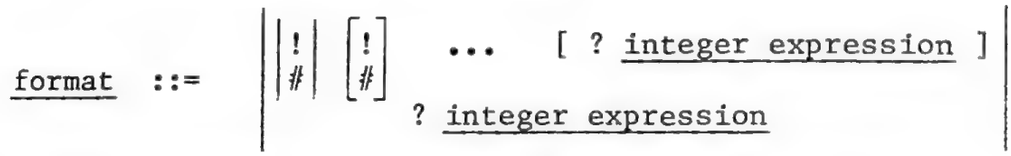
is

Y ␣ A1 ␣ Y ␣ A2 ␣ ... ␣ Y ␣ @ expression atom i ␣ ... ␣ Y ␣ An .

8.1.4 Some Common Elements Among Command Definitions

8.1.4.1 Formats in READ and WRITE

The format, when executed as part of a READ or WRITE command, generates format-controlling output operations. The parameters of a format are executed one at a time, in left-to-right order.



The format-control parameters may take the following forms.

! causes a "new line" operation to be executed at the current device. (See 5.4 for a discussion of the current device.) The effect of ! is equivalent to writing CR LF on a pure ASCII device. In addition CX is set to 0 and 1 is added to CY (see 5.5).

causes a "top of form" operation to be executed at the current device. The effect of # is equivalent to writing CR FF on a pure ASCII device. In addition, CX and CY are set to 0. When the current device is a display, the screen is blanked and the cursor is positioned at the upper-left corner.

? integer expression produces an effect similar to "tab to column integer expression". More precisely, if CX is greater than or equal to the value of integer expression, there is no effect. Otherwise, the effect is the same as writing (integer expression - $X) spaces. ($X is the value of CX.) Note that the leftmost column of a line is given the number 0.

If the current device does not accept output, execution of a format has no effect, except as described on CX and CY.

8.1.4.2 Timeouts in LOCK, OPEN, and READ

The JOB, LOCK, OPEN, and READ commands can make use of an optional timeout specification, invoked by the presence of the timeout in the executed command.

timeout ::= : numeric expression

Each of these three commands has associated with its definition a specific external condition to which the execution of the command is sensitive.

JOB — By definition, execution of JOB is complete when the new process has been initiated. The condition tested is successful initiation of the new MUMPS process.

LOCK — Execution of LOCK may not proceed in violation of the Descendant Exclusivity Rule (see 4.2). The condition tested is the absence of any descendancy relation between each node designator implied by the argument of the LOCK command and the union of Lock Lists over all other partitions of the system.

OPEN — Execution of OPEN may not proceed in violation of the Device Exclusivity Rule (see 4.3). The condition tested is the absence of any common device between any device designator implied by the argument of the OPEN command and the union of Open Lists over all other partitions of the system.

READ — By definition, execution of READ is complete when the input message has been terminated. The condition tested is termination of the input message.

If the optional timeout is absent from the command argument, execution of the command will proceed if and only if the condition associated with the command is satisfied. If the condition is not satisfied, execution will be suspended until the condition is satisfied; then execution will proceed.

If the optional timeout is present in the command argument, a nonnegative value of numeric expression is required. If the value of numeric expression is negative, zero is used. Let t be this nonnegative value; the timeout specifies a suspension of execution of no more than t seconds, as described in the following.

If t = 0, the condition is tested. If it is true, the Test Switch is set to 1; otherwise, the Test Switch is set to 0. Execution proceeds without delay.

If t is positive, execution is suspended until the condition is true, but in any case no longer than t seconds. If at the time of resumption of execution the condition is true, the Test Switch is set to 1; otherwise, the Test Switch is set to 0.

[Port: Programmers should exercise caution in the use of noninteger values for the timeout. The period of actual time which elapses upon encountering a timeout cannot be expected to be exact; relying upon noninteger values in a timeout can lead to unexpected results. :Port]

If the optional timeout is present and the rules stated above require that execution must proceed without the condition having been satisfied, resolution is obtained in the following ways, depending on the command involved.

LOCK — The function specified by the argument in question is only partly executed (that is, the partition's Lock List is left empty) and control proceeds to the next command or argument.

OPEN — The function specified by the argument in question is not executed and control proceeds to the next command or argument.

READ — Whatever input characters have arrived prior to resumption of execution are taken as the whole input message, even if that message is empty. The disposition of those input characters which arrive after resumption of execution but prior to execution of the next READ, if any, is specified by the implementor.

8.2.1 **BREAK**

Syntax

```
BREAK | [ ␣ ]                                          |
      | ␣ argument syntax specified by implementor     |
```

Elaboration

1. The command word may be abbreviated to the single letter B.
2. The command word may be post-conditionalized.
3. The implementor specifies whether there is a form with arguments and, if so, whether arguments may be listed.
4. The implementor specifies whether arguments may be post-conditionalized.
5. The implementor specifies whether argument indirection is permitted.

Execution

1. Argumentless form. No change is made to partition storage or system storage as a direct result of execution of BREAK. Execution is suspended until receipt of a signal, whose nature is specified by the implementor, from a device, whose identity is specified by the implementor. (It is possible that storage values may be changed as a result of external intervention while execution is suspended.)

2. Form with argument. Execution is specified by the implementor.

Examples

1. `B` — Unconditional BREAK

2. `B:X=3` — BREAK if X=3

3. `SET X=^A(3) IF X=0 BREAK  KILL ^A` — A BREAK is inserted before a global is deleted so that the programmer may examine the job's status.

CLOSE 8.2.2

Syntax

CLOSE ␣ device specifier [: device parameters]

device specifier ::= expression

device parameters ::= expression | ([[expression] :] ... expression)

Elaboration

1. The command word may be abbreviated to the single letter C.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The value of the Open List corresponding to this partition is subject to change. Depending on whether the value of the device specifier designates the current device, the content of the Current Device Designator is subject to change.

The value of the expression in the device specifier is a device designator which denotes a device, or it may not, depending on a specification provided by the implementor. If the device designator implied by the argument designates the same device as any device designator in the Open List, then each of the latter is removed from the Open List. See the discussion of device parameters under OPEN (8.2.12).

Cross-reference

4.3 Open List and device designator
5.4 Current Device Designator and Principal Device Convention

Examples

1. C 3 — Device 3 is CLOSEd.
2. C:X=1 X — Device X is CLOSEd only if X=1.

8.2.3 **DO**

Syntax

DO ␣ entry reference

Elaboration

1. The command word may be abbreviated to the single letter D.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of an argument of DO is described in full in terms of the system model in 6.2 paragraph 4. Execution is equivalent to a subroutine call. The entry point of the subroutine is defined by the argument. The exit point occurs upon execution of QUIT not in the scope of FOR, or of eor, neither of which is serving as an exit point for a subsequently executed DO or XECUTE.

Cross-reference

6.2 paragraph 4 Execution description
6.2 paragraph 2 Execution of QUIT not in the scope of FOR, and of eor
Chapter 7 Syntax and interpretation of entry reference

Examples

1.	DO X WRITE Y	The subroutine beginning at the line labeled X is executed; then the variable Y is written.
2.	DO X,Y	This is equivalent to DO X DO Y
3.	DO ^PGM WRITE Y	Routine PGM is invoked; then Y is written.
4.	DO A,^PGM,^ROU	Multiple routine names and labels may be listed as arguments.
5.	DO INT^JTO	Routine JTO is invoked beginning at label INT.
6.	D:X=1 A^PGM,B,^TEST	The list of subroutines is invoked only if X=1.
7.	DO ^CEN:'A,B	Routine CEN is invoked if A=0; subroutine B is invoked independently of the value of A
8.	D @X D @X^PGM D ^@XYZ D @A^@B D:@A=B @C^@D:@E=@F	Indirection may be used in the DO command

8.2.4 **ELSE**

Syntax

ELSE [␣]

Elaboration

1. The command word may be abbreviated to the single letter E.
2. The command word may not be post-conditionalized.

Execution

Execution of ELSE is described in terms of the system model in 6.2 paragraph 3. When ELSE is executed and the Test Switch ($T) has the value 1, the execution of the remainder of the line up to, but not including, the eol is inhibited. In the scope of FOR the effect of each execution of a given ELSE is considered separately at the time it is executed.

Cross-reference

6.2 paragraph 3 Execution description
5.3 Test Switch

Examples

The following two examples produce the same effect.

```
IF X=1 WRITE "YES"
ELSE  WRITE "NO"

WRITE:X=1 "YES" WRITE:'(X=1) "NO"
```

FOR

8.2.5

Syntax

FOR ␣ local variable = for parameter [, for parameter] ...

for parameter ::= | expression | start-step parameter : numeric expression 3 | start-step parameter |

start-step parameter ::= numeric expression 1 : numeric expression 2

Elaboration

1. The command word may be abbreviated to the single letter F.
2. The command word may not be post-conditionalized.
3. Arguments may not be listed; however, for parameters may be listed in any order within the single argument.
4. Arguments may not be post-conditionalized.
5. Argument indirection is not permitted, nor is any indirection permitted at the level of the for parameter.

Execution

The "scope" of this FOR command begins at the next command following this FOR on the same line and ends just prior to the eol on the same line.

FOR specifies and controls repeated execution of its scope for successive values of the named local variable. Each for parameter causes the local variable to be given a specified sequence of values, and the scope to be executed once for each of these values; successive for parameters control this process in turn, in left-to-right order. Any expression occurring in the local variable, for example in a subscript or in indirection, is evaluated once per execution of the complete FOR, prior to the first execution of any for parameter.

The following descriptions specify how each of the for parameter forms specifies and controls the values of the local variable and the sequence of executions of the scope. The variable names A, B, and C are to be considered hidden storage registers used solely for control of each FOR command.

1. If the for parameter is of the form expression.

   a. Set local variable = expression.
   b. Execute the scope once.
   c. Execution of this for parameter is complete.

2. If the for parameter is of the form start-step parameter.

   a. Set A = numeric expression 1.
   b. Set B = numeric expression 2.
   c. Set local variable = A.
   d. Execute the scope once.
   e. Set local variable = local variable + B.
   f. Go to d.

   Note that this procedure specifies an endless loop. Termination of this loop must occur by execution of a QUIT or GOTO within the scope. These two termination methods are available within the scope of any FOR, and they will be described in general below. Note also that no for parameter to the right of one of this form can ever achieve control.

3. If the for parameter is of the form
   start-step parameter : numeric expression 3
   and the step parameter (numeric expression 2)
   has a nonnegative value.

   a. Set A = numeric expression 1.
   b. Set B = numeric expression 2.
   c. Set C = numeric expression 3.
   d. Set local variable = A.
   e. If local variable > C, execution of this for parameter is complete.
   f. Execute the scope once.
   g. If local variable > C-B, execution of this for parameter is complete.
   h. Set local variable = local variable + B.
   i. Go to f.

4. If the for parameter is of the form
start-step parameter : numeric expression 3
and the step parameter (numeric expression 2)
has a negative value.

a. Set A = numeric expression 1.
b. Set B = numeric expression 2.
c. Set C = numeric expression 3.
d. Set local variable = A.
e. If local variable < C, execution of this for parameter is complete.
f. Execute the scope once.
g. If local variable < C-B, execution of this for parameter is complete.
h. Set local variable = local variable + B.
i. Go to f.

If there is more than one FOR in a line, their executions may be considered to be nested, and a FOR to the right is said to be "within" or "inside" a FOR to the left. When two FORs are so nested, one execution of the scope of the outer FOR encompasses one execution of the entire inner FOR command, corresponding to a complete pass through the inner FOR's for parameter list (subject to early termination by a GOTO or QUIT).

Note that any particular execution of any command in the scope of a FOR is under control of exactly one for parameter of that FOR at the time of the command's execution. Two commands, GOTO and QUIT, have special effects when they are executed in the scope of a FOR. These effects are considered below.

Execution of a QUIT within the scope of a FOR has two effects.

a. It terminates this particular execution of the scope at the QUIT; commands to the right of the QUIT are not executed during this execution of the scope.
b. With respect to the innermost FOR in whose scope the QUIT occurs, it causes the remaining values specified by the controlling for parameter, and all values specified by any remaining for parameters in the same for parameter list, not to be calculated, and the scope not to be executed under their control.

In other words, execution of QUIT in the scope of a FOR effects the immediate termination of execution of the innermost such FOR. If there is a next outer FOR, execution of this QUIT effects the immediate termination of one execution of the scope of this FOR.

Execution of a GOTO within the scope of a FOR effects the immediate termination of all FORs to the left of the GOTO in the same line, and it transfers control to the point specified.

Cross-reference

6.2 paragraph 7 Execution of FOR
6.2 paragraph 2 Execution of QUIT in the scope of FOR
6.2 paragraph 5 Execution of GOTO in the scope of FOR
8.2.6 Definition of GOTO
8.2.13 Definition of QUIT

Examples

1. `F I=1:1 WRITE I Q:I>2 W "*"` The output will be 1*2*3

2. `F I=1:1:3 W I` The output will be 123

3. `F I=3:-2:-2 W I` The output will be 31-1

4. `F I=5,3.4,7,9 W I` The output will be 53.479

5. `F I=.4,1:2:5,9,10:1 DO A IF I>15 QUIT`
   A will be executed 12 times

6. `F I=1:1:2 F J=2:2:6 W I,"@",J,"*"`
   The output will be
   1@2*1@4*1@6*2@2*2@4*2@6*

7. `F I=1:1:3 F J=10:10:30 Q:I*J>30 W I,"@",J,"*"`
   The output will be
   1@10*1@20*1@30*2@10*3@10*

8. `F I=3:5:0 WRITE I` Nothing will be written. The final value of I will be 3.

9. `FOR I=.01:.0001:.02 D A` This will execute A 101 times.

10. `F I=X:Y:Z D A`

11. `F I=1:2:10 SET I=I-1 W I` Changing the value of the local variable which is used as an index is permitted. The output will be 0123456789

12. `SET Z=10 F I=2:2:Z SET Z=Z-1 DO A`

    Because the ending value of the loop is computed before the scope is executed, A will be executed 5 times

13. `F I="TEST",X,3:4:5 ...`

    When the simple form of the for parameter is used, the index may be given nonnumeric values.

14. `F I=X:Y:Z F J=A:B:C F K=1:1:3 ... G LCS`

    The GOTO exits from all nested FORs.

15. `F @A=1:1:3 DO B`

    Although indirection is not permitted on the for parameter as a whole, it may be used as usual in place of variable names.

16. `F I=@X:1:3 DO @B`

17. `F @I=@X:@Y:@Z DO @B`

8.2.6 GOTO

Syntax

GOTO ␣ entry reference

Elaboration

1. The command word may be abbreviated to the single letter G.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of an argument of GOTO is described in full in terms of the system model in 6.2 paragraph 5. Execution is equivalent to a simple transfer of control. If the executed argument names another routine, the routine body at the top of the partition stack is replaced by the named routine body. Note that, in contrast to DO, when arguments are listed, at most one argument actually transfers control. When GOTO is executed in the scope of FOR, all FORs to the left of the GOTO in the same line are terminated with the transfer of control.

Cross-reference

6.2 paragraph 5 Execution description
Chapter 7 Syntax and interpretation of entry reference
8.2.5 Execution of GOTO in the scope of FOR

Examples

1.	GOTO XYZ	Execution continues at the start of the line labeled XYZ in this routine.
2.	GOTO ^PGM	Context switches from the routine containing the GOTO to the first line of the routine PGM.
3.	G CC+2^N1H	Context switches to routine N1H and execution begins at the second line after the line labeled CC.
4.	G 13^GRAY:A=1	If and only if A=1, context switches to routine GRAY and execution begins at the line labeled 13.
5.	G:A=1 13^GRAY	Equivalent to Example 4.
6.	G ^ADM:A=0,X^CEN	If A=0, context switches to routine ADM and execution begins at its first line; otherwise, context switches to routine CEN and execution begins at the line labeled X.
7.	G:A=1 AAA:B=2,13:D=4	This line is equivalent to G AAA:A=1&(B=2) G 13:D=4&(A=1) .
8.	G @X G ^@B G A^@B G @A^B G @A^@B	There are many opportunities to use indirection within the GOTO command
9.	G:X=@Y @A+@C+D^@PGM:@E=@F	If control transfers, the line at which execution continues is the @C+Dth line after label @A .

8.2.7 HALT

Syntax

HALT [␣]

Elaboration

1. The command word may be abbreviated to the single letter H. Note that when HALT is abbreviated, the absence of an argument distinguishes it from HANG.
2. The command word may be post-conditionalized.

Execution

Execution of the process in this partition terminates. System storage associated with this partition is left in the following state.

1. The element of the Job Number P-vector corresponding to this partition is zero.
2. The element of the Lock List P-vector corresponding to this partition is empty.
3. The element of the Open List P-vector corresponding to this partition is empty.

Cross-reference

6.2 paragraph 1 Execution of HALT

Examples

1.	H	Unconditional HALT
2.	H:A=1 WRITE "TEST"	If A=1, the process HALTs. Otherwise, execution continues and TEST is written.
3.	H:A=@B	Indirection may be used in the post-conditional.

HANG 8.2.8

Syntax

HANG␣ numeric expression

Elaboration

1. The command word may be abbreviated to the single letter H.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

Let t be the numeric interpretation of the value of the argument. If t is zero or negative, HANG is of no effect. If t is positive, execution is suspended for t seconds. No storage contents are changed as a direct result of execution of HANG. [Port: Programmers should exercise caution in the use of noninteger values for the HANG command. The period of actual time which elapses upon execution of a HANG command cannot be expected to be exact; relying upon noninteger values in the HANG command can lead to unexpected results. :Port]

Cross-reference

Examples

1. `A IF $P($H,",",2)<X HANG 3 G A`
   If the number of seconds after midnight is less than X, execution hangs three seconds and the line repeats.

2. `H:I=3 10 WRITE !`
   If I=3, a ten-second HANG is executed before the WRITE command is executed.

3. `H @X`
   `H:$E(@A,B,@C) X+@Y+Z`
   Indirection may be used in the HANG command.

4. `H 7.5`
   Noninteger values can be used in the HANG command.

8.2.9 IF

Syntax

Argumentless form: IF [␣]

Form with argument: IF ␣ truth-value expression

Elaboration

1. The command word may be abbreviated to the single letter I.
2. The command word may not be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of both forms of IF may be viewed in a unified way as follows. Execution consists of two steps. The first step is not executed by the argumentless form. The second step is common to both forms.

Step 1. The truth-value interpretation of the value of the argument is placed into the Test Switch.

Step 2. If the content of the Test Switch is 1, there is no action; that is, execution continues at the next command or argument. If the content of the Test Switch is 0, execution of every character in this line to the right of this argument, up to but not including the eol, is inhibited. There can be no direct or side effect arising from any such inhibited execution.

Cross-reference

6.2 paragraph 3 Execution of IF and ELSE
8.2.4 Execution of ELSE
8.1.3.2 Difference between post-conditional and IF

Examples

1. `I X=1 WRITE X`

   The code to the right of the IF command is executed only if X=1.

2. `I Y S Z=3`

   The code to the right of the IF command is executed only if the numeric interpretation of the value of Y is nonzero.

3. `I X=1,Y=2 DO 3`

   Either of the two following forms will or will not DO 3 under the same conditions.
   `IF X=1 IF Y=2 DO 3`
   `IF X=1&(Y=2) DO 3`
   That is, the effect on the sequence of execution of listing arguments is similar to an implied and.

4. `I X,^A(Y) G 3`

   The side effects, however, may be different. This code is not equivalent to
   `I X&(^A(Y)) G 3` since the latter will always evaluate ^A(Y) while the former might not.

5. `I X?1N2A!(Y=2)`

   Complex expressions occur frequently in IF commands

6. ```
   IF X=1 W !,"TEST"
   ELSE  W !,"EXAM"
   IF  W " ARGUMENTLESS"
   ```

   If X=1, this code will write
   TEST ARGUMENTLESS
   otherwise, it will write EXAM.

7. ```
   IF @X
   IF A,@B,C
   IF A=@B,@C=D,E=F+@G+H
   ```

   Indirection may be used in the IF command

JOB

Syntax

JOB␣job specifier [: job parameters [timeout]]
                   [: timeout]

job specifier ::= entry reference

job parameters ::= | expression |
                   | ([[expression] :] ... expression) |

Elaboration

1. The command word may be abbreviated to the single letter J.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The JOB command has the effect of initiating a distinct MUMPS process, with its own unique Job Number, and making it active by implicitly executing a DO command with the named entry reference as its argument. Complete discussions of the actions taken can be found in 4.4 and in 6.2 paragraph 4. The MUMPS process which executes the JOB command continues execution when the implicit DO command is initiated for the process begun by the JOB command. The two processes then operate in parallel as distinct entities.

If there is no timeout in the argument, execution will be suspended only until the new MUMPS process is initiated. As soon as the new MUMPS process is begun, execution will proceed. The Test Switch will not be affected.

If there is a timeout in the argument, the definition of execution is the same as that stated above, with two additional elements. Let t be the nonnegative value implied by the timeout (see 8.1.4.2).

1. If $t \neq 0$, no suspension of execution occurs. If t is positive, execution is suspended if and only if necessary as above to initiate the new MUMPS process, but at the end of t seconds, resumption of execution is forced. (In the following, if t=0, the "time of resumption of execution" is defined to be the time at which execution occurs.)

2. At the time of resumption of execution, the condition of whether or not the new MUMPS process was successfully initiated is tested. If it is true, the Test Switch is given the value 1 and execution proceeds. If it is false, the Test Switch is given the value 0 and execution proceeds.

The job parameters portion of the argument is used to specify whatever parameters the implementor chooses to associate with initiation of the process in question. The permissible syntax of the value of each expression in the job parameters, and the interpretation of each value, is specified by the implementor.

[Port: MUMPS routines designed for portability may not make use of the JOB command. :Port]

Cross-reference

Chapter 3		MUMPS system model
	4.4	Job Number P-vector
	7	Syntax and interpretation of entry reference

Examples

1. `JOB X`

   A new MUMPS process is initiated with the routine of this MUMPS process, and execution begins at label X of the routine.

2. `JOB ^ABC,^GHI,^XYZ`

   In turn, a new MUMPS process is initiated and execution begins at the first line of the routines ABC, GHI, and XYZ respectively. A total of three new MUMPS processes are initiated by this command, which is equivalent to the sequence:
   `JOB ^ABC JOB ^GHI JOB ^XYZ`

3. `JOB ^NAME::10 G ERR:'$T`

   This process attempts to initiate a new process with execution beginning at the first line of routine NAME for up to 10 seconds. If it is successful, $T will assume the value 1. Otherwise, $T will be 0 and execution will continue at the line labelled ERR.

4. `J ^NAME::10 ELSE G ERR`

   This is equivalent to the above.

5. `J:N>6 ^BODY`

   The JOB command may be post conditionalized.

6. `J NEW^ORDER:("PORD":2:150:"\SFO"):TIM`

   In some implementations, additional arguments can be used for setting job-specific parameters.

7. `J @X`

   Arguments of JOB may be indirected.

8.2.10 **KILL**

Syntax

1. "Kill All" argumentless form: KILL [␣]
2. "Selective Kill" form: KILL ␣ storage reference
3. "Exclusive Kill" form: KILL ␣ (name [, name] ...)

Elaboration

1. The command word may be abbreviated to the single letter K.
2. The command word may be post-conditionalized.
3. Arguments may be listed. In an argument list the two argument forms may be mixed. (Note that the Exclusive Kill syntax form contains only one argument.)
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The execution of all three forms may be defined in terms of a single atomic operation: the killing of a single variable. Let V be a storage reference. Let N be the corresponding node, if any, in Named Partition Storage or Named System Storage, respectively, for which the Name attribute corresponds to V. (See 12.4 for a discussion of this correspondence.)

Killing V has the following effect. If N does not exist, there is no effect. If N exists, N and every descendant of N is deleted. No attribute of any ascendant of N is changed when V is killed.

1. Execution of a Kill All kills all local variables.
2. Execution of a Selective Kill kills the variable named in the argument.
3. In the Exclusive Kill form, name denotes an occurrence of local variable containing no subscripts. All unsubscripted local variables except for the one or more named in the argument are killed. Consequently, all descendants of all local variables are killed except for the descendants of the variables named in the argument.

Cross-reference

4.1 Structure of Named System Storage
5.1 Structure of Named Partition Storage
12.4 storage reference

Examples

1.	KILL X	Local variable X is killed.
2.	KILL X,Y,^A(3,4)	Local variables X and Y, and global variable ^A(3,4) are killed.
3.	KILL (A)	All local variables except A and its descendants are killed.
4.	K (A,C,DEF,%1)	All local variables except those named and their descendants are killed.
5.	K SET X=$T	The argumentless kill causes all local variables to be killed. After execution of this line, X will be the only local variable with a defined value (provided that $T is defined).
6.	K:X=1 ^A(X)	If X=1, then ^A(1) is killed. All forms of KILL may have a post-conditional on the command word.
7.	K ^A	The entire global array ^A is killed.
8.	K @X K A,@B,C K:@B=C A(B,@C)	Indirection may be used in the KILL command

8.2.11 **LOCK**

Syntax

1. Argumentless form:

   LOCK [␣]

2. Form with argument:

   LOCK ␣ | variable name | (variable name [, variable name] ...) | [timeout]

Elaboration

1. The command word may be abbreviated to the single letter L.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The upper argument form, LOCK ␣ variable name [timeout] , is equivalent to the lower argument form in which exactly one name occurs: LOCK ␣ (variable name) [timeout] . Therefore the following definition of the form with argument deals only with the second, more general, argument form.

Execution of both forms of LOCK may be viewed in a unified way as follows. Execution consists of two steps. The first step is common to both forms. The second step is not executed by the argumentless form.

Step 1. The element of the Lock List P-vector corresponding to this partition is unconditionally made empty.

Step 2. Subject to the conditions described below, the element of the Lock List P-vector corresponding to this partition is given a new value. The following comments apply to this process.

In 8.1.4.2 it was stated that the presence of a timeout in a command argument denotes that execution is dependent on an external condition whose definition is associated with the particular command. The condition associated with LOCK is true if and only if the Descendant Exclusivity Rule (referred to in the following as "the Rule", see 4.2) would not be violated by replacement of the now-empty Lock List associated with this partition by a new Lock List denoted by the argument. Stated another way, the Rule mandates a particular kind of noninterference which must hold between all possible pairs of partition Lock Lists. This LOCK argument defines a new Lock List (in a way to be described below) for this partition. The new Lock List, taken as a whole, must not violate the Rule. If indeed it would not, the condition is true; if it would violate the Rule, the condition is false. Of course, the truth of the condition can be a function of time, since other partitions' Lock Lists may be changing.

If the timeout is absent, execution is suspended, if and only if necessary to satisfy the Rule, until assignment of the new value to this partition's Lock List can occur without violation of the Rule. When the Lock List can be given the entire new value, it will, and execution will proceed. The Test Switch will not be altered.

If the timeout is present, the definition of execution is the same as that stated above, with two additional elements. Let t be the nonnegative value implied by the timeout (see 8.1.4.2).

1. If t=0, no suspension of execution occurs. If t is positive, execution is suspended if and only if necessary as specified above, but at the end of t seconds, resumption of execution is forced. (In the following, if t=0, the "time of resumption of execution" is defined to be the time at which execution occurs.)

2. At the time of resumption of execution, the condition is tested. If it is true, the Lock List is given its designated value, the Test Switch is given the value 1, and execution resumes. If the condition is false, the Lock List remains empty, the Test Switch is given the value 0, and execution resumes.

There are no partial assignments to the Lock List of a sublist of the list specified by an argument, and there are no retroactive or delayed changes to a Lock List. If a process fails to establish its Lock List as desired upon execution of a LOCK, any future retries must be explicitly executed by means of the LOCK command.

The Lock List specified by the argument contains as many elements as the argument contains variable names, and they are in one-to-one correspondence. For each variable name in the argument, the corresponding Lock List element is the node designator derived from that variable name. See 12.4 for the syntax of variable name and the algorithm for deriving its implied node designator.

Since an argument of LOCK is only a name and not a reference to Named Partition or System Storage, the following propositions hold.

1. There is no relationship between a variable name appearing as an argument of LOCK and the existence of, or the attributes of, any node in Named Partition or System Storage.
2. The appearance of a global variable as an argument of LOCK does not affect the Naked Indicator.

Cross-reference

4.2 Lock List in System Storage
5.3 Test Switch
8.1.4.2 timeout
12.4 Node Designation Mapping, variable name

Examples

1. LOCK ^A

Between the time execution is resumed after this LOCK and the time of execution of the next LOCK by this process, no other process will be able to execute an argument of LOCK containing the unsubscripted global name ^A or any subscripted global name whose name part is A. In addition, any variable names previously LOCKed by this process are unLOCKed.

2. L (^B(1,2),^C(3))

Between the time execution is resumed after this LOCK and the time of execution of the next LOCK in this partition, no other process will be able to LOCK any of the following variables.
^B ^B(1) ^B(1,2,3)
^C ^C(3) ^C(3,4,5)
However, another process can LOCK any of the following.
^B(2) ^B(1,3,4)
^C(4)

3.	`L  SET X=1`	The argumentless LOCK command releases the effect of any previous LOCK in this partition.
4.	`L D(1):3 G A:'$T`	An attempt will be made for an interval of three seconds to LOCK D(1). If the attempt is successful, $T will assume a value of 1. If unsuccessful, $T will be zero and execution will continue at the line labeled A.
5.	`L D(1):3 ELSE  G A`	This has the same effect as the example above.
6.	`L:X=1 ^EF(^A(5))`	The LOCK command can be post-conditionalized. Appearance of a global variable name does not affect the Naked Indicator unless, as in this example, it occurs in an evaluated expression. Here it is a subscript; other such possibilities are in post-conditionals and indirection.
7.	`L @A` `L:@B=C X(@A),@B`	Indirection may be used in the LOCK command.

8.2.12 **OPEN**

Syntax

OPEN ␣ device specifier [: device parameters [timeout] | timeout]

device specifier ::= expression

device parameters ::= expression | ([[expression] :] ... expression)

Elaboration

1. The command word may be abbreviated to the single letter O.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The device specifier in the command argument denotes an input-output "device", which is a sequential character source and/or sink. The value of this expression is to be used as a device designator in the Open List corresponding to this partition, as described in 4.2. The implementor specifies the syntax rules which the value of this expression must satisfy. Also, the implementor specifies the correspondence between the possible values of this expression and the set of available devices.

A partition is said to "own" device D if and only if a device designator designating D is in the partition's Open List.

The intended function of OPEN with one argument is to obtain ownership of one device. Execution of each argument of a multiple-argument form of OPEN will add, in turn, each device designator specified by the argument to the partition's Open List unless doing so would violate the Device Exclusivity Rule ("the Rule"). The Rule states that no two partitions may own the same device. The condition associated with the OPEN command (see 8.1.4.2) is true if and only if adding the specified device designator to the partition's Open List would not violate the Rule.

If there is no timeout in the argument, execution will be suspended if and only if necessary to avoid violation of the Rule. When the device designator specified by the argument may be added to the Open List associated with this partition without violating the Rule, it will be, and execution will proceed. The Test Switch will not be affected.

If there is a timeout in the argument, the definition of execution is the same as that stated above, with two additional elements. Let t be the nonnegative value implied by the timeout (see 8.1.4.2).

1. If t=0, no suspension of execution occurs. If t is positive, execution is suspended if and only if necessary as specified above, but at the end of t seconds, resumption of execution is forced. (In the following, if t=0, the "time of resumption of execution" is defined to be the time at which execution occurs.)

2. At the time of resumption of execution, the condition associated with the OPEN command is tested. If it is true, the device designator is added to the Open List, the Test Switch is given the value 1, the device parameters are disposed of as specified below, and execution proceeds. If the condition is false, the Open List remains unchanged, the Test Switch is given the value 0, there is no effect due to the device parameters, and execution proceeds.

The device parameters portion of the argument is used to specify whatever parameters the implementor chooses to associate with the device in question. The permissible syntax of the value of each expression in the device parameters, and the interpretation of each value, is specified by the implementor. This much, however, is common among all implementations. Device parameters are uniquely stored in association with each device, not each partition. At system initiation, each device is given an appropriate set of default parameters, as specified by the implementor. Each time an OPEN, CLOSE, or USE is executed, one or more device parameters may be changed by the executed argument. The value of each device parameter persists until it is replaced by a subsequent successful execution of an argument of OPEN, CLOSE, or USE in any partition which specifies a value for the device parameter.

Cross-reference

4.3 device designator

Examples

1.	OPEN 4	Ownership of device 4 is obtained. If another process already owns device 4, this process will hang until the device is available.
2.	O 4,X,Y	Similarly, devices 4, X, Y are OPENed in turn.
3.	O X::3 G A:'$T	The process attempts to OPEN device X for up to three seconds. If it is successful, $T will assume a value of 1. Otherwise, $T will be zero and execution will continue at the line labeled A.
4.	O X::3 ELSE G A	This is equivalent to the above.
5.	O:X=1 A,B,C	The OPEN command word may be post-conditionalized.
6.	O A:("IO":300:"TTY"),B:("O":800:"MT":20):TIM	In some implementations, additional arguments may be used for setting device-specific parameters.
7.	O @X	Indirection can be used in arguments of the OPEN command.

QUIT

8.2.13

Syntax

QUIT [␣]

Elaboration

1. The command word may be abbreviated to the single letter Q.
2. The command word may be post-conditionalized.

Execution

QUIT may occur in either of two distinct contexts.

1. In the scope of FOR.
2. Not in the scope of FOR.

The function of execution of QUIT not in the scope of FOR is to cause an exit from the subroutine initiated by a DO or XECUTE argument. (The initial activation of this process is assumed to be the result of an externally executed DO.) In terms of the system model, execution of QUIT not in the scope of FOR pops the top level off the Partition Stack (see 6.2 paragraph 2). If this leaves the stack empty, HALT is executed.

Execution of eor, i.e., executing off the end of a routine body, is equivalent to execution of QUIT not in the scope of FOR.

Execution of QUIT in the scope of FOR immediately terminates execution of the innermost FOR in whose scope the QUIT occurs. If the line containing the QUIT has only one FOR, execution of the QUIT terminates execution of the line. If the line has more than one FOR, execution of the QUIT causes termination of the innermost FOR and causes termination of the current execution of the scope of the second-from-innermost FOR.

Cross-reference

6.2 paragraph 2 Execution of QUIT
8.2.3 DO
8.2.5 FOR
8.2.7 HALT
8.2.19 XECUTE

Examples

1. `I X>3 Q  WRITE Y`

2. `Q:X>3  WRITE Y`

   These two examples do not have the same effect because the scope of the IF extends to the end of the line. In the first example the WRITE Y is never executed.

3. Consider the following routine body.

```
A F I=1:1:100 D B Q:I+X>6  W I
  W " END" Q
B S X=I*2 D C W "B"
  Q
C W "C" Q
```

In this routine body:

   a. The QUIT in line A terminates the FOR loop.
   b. The QUIT in line A+1 terminates the routine.
   c. The QUIT in line B+1 ends execution of the DO B causing execution to return inside the FOR loop.
   d. The QUIT in line C ends execution of the DO C causing execution to return to line B.

The output is CB1CB2CB END

READ

8.2.14

Syntax

```
READ ␣ | string literal                          |
       | format                                  |
       | local variable [ timeout ]              |
       |*local variable [ timeout ]              |
```

Elaboration

1. The command word may be abbreviated to the single letter R.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of READ causes input and/or output on the current device. Each argument form governs the execution of READ according to a different set of rules.

READ string literal causes output of the string literal to the current device, with alteration of the values of CX and CY as described in 5.5. If the current device does not accept output, the effect on CX and CY is still as described in 5.5, but no output operation is performed.

READ format causes output of format control information to the current device, with alteration of the values of CX and CY as described in 8.1.4.1. If the current device does not accept output, the effect on CX and CY is still as described in 8.1.4.1, but no output operation is performed.

READ local variable [timeout] causes an input ASCII data string from the current device to be accumulated until a termination criterion is met. This termination criterion is dependent on whether the timeout is present, as described below.

The implementor defines an "explicit termination procedure" which may be device-dependent. If the explicit termination procedure involves entry of a character, the implementor defines whether or not that character becomes part of the input string when the procedure is executed.

Let t be the nonnegative value associated with the timeout. (See 8.1.4.2.)

1. If the timeout is absent, execution is immediately suspended until the explicit termination procedure is executed; then execution immediately resumes. The input string is the concatenation of all characters received during the time that execution is suspended, with characters received earlier at the left.

2. If the timeout is present and t=0, execution is not suspended and the input string is empty.

3. If the timeout is present and t is positive, execution is immediately suspended. Execution resumes either at the end of t seconds or immediately upon explicit termination, whichever is earlier. The value of the input string is the concatenation of all characters received during the time that execution is suspended, with characters received earlier at the left.

Let I be the value of the input string as defined above. At the time of resumption of execution, the following occurs.

1. Execute SET local variable = I.

2. If the timeout is absent, the Test Switch is unchanged. If the timeout is present and t=0, the Test Switch is given the value 0. If the timeout is present and t is positive and the explicit termination procedure did not occur at or prior to resumption of execution, the Test Switch is given the value 0. If the timeout is present and t is positive and the explicit termination procedure occurred at or prior to the resumption of execution, the Test Switch is given the value 1.

READ *local variable [timeout] is a one-character read, where the input character may be an ASCII character, or it need not be; for example, the input "character" may be a status value.

The local variable named in the argument always receives an integer value. Whether the integer value is associated with a character in a code table (for example, it may be the decimal equivalent of the binary ASCII code of the input character), or whether it has some other interpretation, is a matter which the implementor may define in a device-dependent way.

1. If the timeout is absent, execution is immediately suspended until one input character arrives from the current device; then execution immediately resumes.

2. If the timeout is present and t=0, execution is not suspended and no character is considered to have arrived from the current device.

3. If the timeout is present and t is positive, execution is immediately suspended. Execution resumes either at the end of t seconds or immediately upon receipt of one character from the current device, whichever is earlier.

At the time of resumption of execution, the following occurs.

1. If a timeout is present and either t=0 or t is positive and no character arrived during suspension of execution, execute SET local variable = -1.

2. Otherwise, execute SET local variable = the integer value associated with the input character.

3. If the timeout is absent, the Test Switch is unchanged. If the timeout is present and t=0, the Test Switch is given the value 0. If the timeout is present and t is positive and no input character arrived during suspension of execution, the Test Switch is given the value 0. If the timeout is present and t is positive and an input character arrived during suspension of execution, the Test Switch is given the value 1.

The form READ local variable [timeout] (that is, the form "without asterisk") affects CX and CY as described in 5.5. The form with asterisk may also affect CX and CY according to 5.5, depending on the implementor's interpretation of the input character.

A strict reading of the execution definition of the last two forms of READ leads to the conclusion that any character or explicit termination procedure arriving prior to the execution of this READ and, if there was a prior READ, after its execution, will be lost. This interpretation may be undesirable in certain operating environments. Therefore, the implementor has latitude to make some or all of the following re-interpretations, in a consistent, but possibly device-dependent, way.

1. Input characters are queued and received from the current device by the READ command in first-in, first-out order in such a way that no character is lost. In the case of READ without asterisk, all characters arriving from the current device prior to execution of this READ and after all prior READs are concatenated to the left of the input string described above. In the case of READ with asterisk, the first character to arrive after all prior READs is the one processed.

2. In the case of READ without asterisk, the implementor must decide how to treat multiple explicit terminations. There are two choices.
   a. Terminations are queued with, and partition, input characters, and each READ accepts only one such partition.
   b. If, at the start of the READ, a character has been received since the most recent termination, all prior terminations are ignored. If a prior termination has not been followed by a character, then that termination is considered to occur immediately after the start of the READ.

3. When a timeout is present, the case t=0 ceases to be a special case and may, along with positive values of t, give a value of 1 to the Test Switch if the condition being tested (explicit termination or arrival of a character, respectively) occurred prior to this READ and subsequent to all prior READs. However, assignment to the Test Switch of the value 1 means, and only means, that the value given to the named local variable is associated with a specific condition. In the case of READ without asterisk, the explicit termination procedure was executed. In the case of READ with asterisk, the value assigned to the local variable is associated with an input character and is not the result of the default assignment of the value -1 to the variable.

Cross-reference

8.1.4.1 format
8.1.4.2 timeout
12.2 string literal
12.4 local variable

Examples

1. READ X

   A data string is entered into local variable X.

2. R X,Y,Z

3. R "PATIENT?",X

   The message PATIENT? appears on the current device; then execution waits for data to be entered, after which it is placed into X.

4. R !!,"FIRST?",X,?30,"SECOND?",Y

   This code causes two carriage return-line feeds to occur on the current device. Then the message FIRST? is displayed. After data is entered into variable X, the device spaces to column 30 and the message SECOND? is displayed. Data may then be entered and read into variable Y.

5. R "TIMING?",X:10 G A:'$T
   R "TIMING?",X:10 E G A

   These two lines do the same thing. After displaying TIMING? the job will hang for ten seconds waiting for input. If the return key (the assumed termination procedure) is pushed during that time period, $T will be set to 1. Otherwise, $T will be 0, X may contain partial input, and execution will continue at A. If no characters have been entered, X will contain the empty string.

6. `R *X,*Y,*Z`

   Under one interpretation available to the implementor, this form of READ may be used for entry of single characters and conversion to their ASCII-equivalent numeric codes. If the characters AB CR were entered, the following values would result.
   X=65
   Y=66
   Z=13

7. `R *X:10 E  G A`

   The asterisk form may also use a timeout. If the timeout expires without a character having been entered, X will have the value -1 and execution will continue at A.

8. `R:$D(^A) !!,"TEST",X:12,*Y,*Z:10,#,"SAMPLE",!,A:9`

   The READ command may be post-conditionalized. All of the various argument forms may appear in one argument list. When more than one timeout is present, the value of $T reflects the last-executed (rightmost) timeout.

9. `R @X`
   `R X:@Y`
   `R:@C @X:@Y`
   `R *@A`

   Indirection can be used with READ.

SET

8.2.15

Syntax

SET ␣ | storage reference | (storage reference [,storage reference] ...) | = expression

Elaboration

1. The command word may be abbreviated to the single letter S.
2. The command word may be post-conditionalized.
3. Arguments may be listed. (The form shown above is a single argument.)
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The SET command provides the means whereby a data string is placed into the value attribute of one (or more) node(s) of Named Partition or System Storage. The data string is the value of the expression; the node(s) is (are) that (those) designated by the storage reference(s). (The words in parentheses apply to the parenthesized form of the argument.)

Execution occurs in the following order.

1. Any indirections or subscripts in the storage reference(s) are evaluated, in left-to-right order as they appear in the argument.
2. The expression is evaluated. Let V denote the value of the expression.
3. For each storage reference (one at a time in left-to-right order if there are several), V is placed into the Value attribute of the node of Named Partition Storage or Named System Storage denoted by the storage reference. (The details of this mapping from a storage reference to a node designator are discussed in 12.4.) This may involve the following considerations.

    a. If the storage reference is a naked reference, the content of the Naked Indicator is used to produce the node designator, and then this resulting node designator causes the content of the Naked Indicator to be given a (possibly new) value. The process is described in 5.2 and 12.4.

b. If there does not exist a node with a Name attribute as denoted by the storage reference, a minimum number of nodes is created so that
   (1) A node with the Name attribute defined by the storage reference exists, and
   (2) It is a descendant of the directory node in either Named System Storage (if the first character of the Name attribute is "^") or Named Partition Storage (otherwise).

c. If any nodes are so created, their Name attributes are defined by the description of the creation process given above. Their other attributes are defined as follows.
   (1) The node whose Name attribute is defined by the storage reference is given the D attribute 1 and the Value attribute V.
   (2) Any other nodes so created (and they form an unbroken chain of ascendants of the above node beginning with its immediate ascendant) are given the D attribute 10. Their value attributes, being unavailable, need not be given any value.

d. Also, if any node is so created, there exists a unique node which, prior to the creation had no descendant and after the creation has all of the created nodes as descendants. The D attribute of this node is given the following value.

If it was	then	it becomes
0		10
1		11
10		10
11		11

e. If the node whose Name attribute denoted by the storage reference does exist, V is placed into its Value attribute and its D attribute is given the following value.

If it was	then	it becomes
0		1
1		1
10		11
11		11

Cross-reference

4.1 Named System Storage
5.1 Named Partition Storage
5.2 Naked Indicator
12.4 storage reference and the mapping to a node designator

Examples

1. SET ^X=Y

   The content of the Value attribute of the level 1 Partition node named Y (i.e., the "value of the unsubscripted local variable Y") is placed into the Value attribute of the level 1 System node (i.e., the "unsubscripted global variable") named ^X.

2. S ^A(1,2)=B(3,4)

   Similarly for subscripted variables.

3. S ^A(1,2)=X,B(4)=7,^C="TEST",X=^D

   Multiple arguments of SET.

4. S ^(2)=^C(X,2)+1

   Because the expression to the right of the = is evaluated before the variable name to its left is determined, the naked reference ^(2) denotes ^C(X,2). If the line had been

   ```
   S ^C(X,2)=^(2)+1
   ```

   the meaning of the naked reference would depend on the state of the Naked Indicator prior to execution of the SET.

5. S (A,B,C)=X

   This is the "multiple" set. All variables in the parentheses are set to the same value.

6. S (A,B,C,D)=0,^A(3)=W,(^A(3,4),^B(5))=I

   Multiple and single SETs may be mixed in one command.

7. S ^A(5)=10,B(^(3))=^C(4)

   When subscripts occur to the left of the =, they are evaluated before the right-side expression. Thus this line is equivalent to

   ```
   S ^A(5)=10,B(^A(3))=^C(4)
   ```

8. `S ^A(3)=10,^(^(3))=^C(4)`

This line is executed as follows.
a. ^A(3) is given the value 10.
b. The subscript ^(3), which is ^A(3), is evaluated as 10.
c. ^C(4) is evaluated.
d. This value is given to ^(10), which is ^C(10).
This line is therefore the same as
`S ^A(3)=10,^C(^A(3))=^C(4)` .
The order is always:
a. First the left-hand subscripts.
b. Then the right-hand expression.
c. Then the left-hand name.

9. `S I=3,(I,A(I))=4`

Even in a multiple SET, the left-hand subscripts are evaluated before the assignment is made. But each argument is completed before the next is begun. Thus, this line is equivalent to
`S I=3,(I,A(3)))=4`
and I ends up with the value 4.

10. `S:X=3 X=4.5`

The SET command may be post-conditionalized. In this example, if X has the value 3, then it is given the value 4.5.

11. `S X=$S(X=3:4.5,1:X)`

The $SELECT function is valuable when used in the right-hand expression of a SET command. This line has the same effect as the previous example.

12. `S X=X=3*1.5+X`

The expression to be evaluated may be quite complex. In this example, the expression X=3*1.5+X is evaluated, and the result is given to the variable X. This line has the same effect as the previous two examples.

13.
```
S @X
S @A=B+C
S A=@B+C
S A=B+@(C_D)
```

Indirection is often used in the SET command.

USE

8.2.16

Syntax

USE ␣ device specifier [: device parameters]

device specifier ::= expression

device parameters ::= expression | ([[expression] :] ... expression)

Elaboration

1. The command word may be abbreviated to the single letter U.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

The device specifier in the command argument denotes an input/output "device", which is a sequential character source and/or sink. The value of this expression is to be used as a device designator in the partition's Current Device Designator, as described in 5.4.

The intended function of USE is to make the designated device the current device by moving the value of the device specifier into the Current Device Designator. Execution proceeds as follows.

1. If, and only if, the content of the Current Device Designator designates a device, the values of CX and CY are placed into the X and Y registers specific to that device.

2. One of the following two alternatives exists.

   a. If the value of the device specifier designates a device and if there is a device designator in the partition's Open List which designates the same device, then the value of the device specifier is placed into the Current Device Designator, the X and Y values specific to that device are placed into CX and CY, respectively, and the device parameters are recorded as described under OPEN (8.2.12). (Translation: The device designated by the USE command is made the current device of the process executing it.)
   b. Otherwise, an erroneous condition exists.

The value of the Current Device Designator may be obtained by execution of the special variable $IO as an expression atom.

Cross-reference

4.3 device designator
5.4 Current Device Designator
8.2.12 device parameters
Chapter 9 $IO special variable

Examples

1.	USE 3	Device 3 is specified for routing of READ and WRITE data.
2.	U 3 SET X=$I	X will receive the value 3.
3.	U X:Y	In some implementations, Y may be used to specify device parameters.
4.	U:X'=0 X	The USE command may be post-conditionalized. This example inhibits making device 0 the current device.
5.	U @X	Argument indirection is permitted.

VIEW

8.2.17

Syntax

VIEW argument syntax specified by implementor

Elaboration

View is an implementation-specific command made available to implementors who may wish to output data not otherwise available. Whether or not it takes one or more arguments, and what any argument syntax is, are specified by the implementor. Each implementation must recognize and accept the VIEW command regardless of any interpretation given to it.

[Port: Routines designed for portability should not contain the VIEW command. :Port]

8.2.18 **WRITE**

Syntax

```
WRITE ␣ | format
        | expression
        | *integer expression |
```

Elaboration

1. The command word may be abbreviated to the single letter W.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may not be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of an argument of WRITE causes output of data and/or control information to the current device. Each argument form governs the execution of WRITE according to a different set of rules.

WRITE format causes output of format control information to the current device, with alteration of the values of CX and CY as described in 5.5 and 8.1.4.1. If the current device does not accept output, the effect on CX and CY is still as described there, but no output operation is performed.

WRITE expression causes output of the value of the expression, one character at a time, in left-to-right order. The effect of each character at the device is defined by the ASCII standard and conventions. Each character of the output affects the values of CX and CY as described in 5.5. If the current device does not accept output, the effect on CX and CY is still as described in 5.5, but no output operation is performed.

WRITE *integer expression is a one-character write whose possibly device-dependent interpretation is defined by the implementor. Possible interpretations include the following.

1. A device command, such as a tape-unit rewind.
2. A numeric device parameter, such as a disk arm position or an absolute plotter coordinate.
3. Any ASCII character the decimal equivalent of whose binary code is the value of the integer expression.

The implementor defines the effect of this form of WRITE on CX and CY.

It is intended that, except for the effects of timeouts, READ and WRITE have identical side effects for identical data transfers.

Cross-reference

5.4 Current device
5.5 CX and CY
8.1.4.1 format
8.2.14 READ

Examples

1. WRITE X

2. W !,"Line feed",#,"Form feed",?20,"TAB",A(3)
   The arguments of the WRITE command may be a mixed selection.

3. W *7,*7,*97
   Under interpretation 3 above, this command would ring the bell twice, then output an "a".

4. W X,*Y,*Z,!!,"TEST",A=B
   Expressions containing operators may be used as arguments.

5. W:X>10 $J(X,7,2)
   The WRITE command may be post-conditionalized. The $JUSTIFY function is frequently used in arguments of WRITE.

6. W @X
   W Z,@X,Y
   Indirection may be used in arguments of WRITE.

8.2.19 **XECUTE**

Syntax

XECUTE _ expression

Elaboration

1. The command word may be abbreviated to the single letter X.
2. The command word may be post-conditionalized.
3. Arguments may be listed.
4. Arguments may be post-conditionalized.
5. Argument indirection is permitted.

Execution

Execution of an argument of XECUTE is described in full in terms of the system model in 6.2 paragraph 6. Its effect is to call a one-line subroutine whose spelling is the value of the expression, preceded by ls and followed by eol. Control automatically returns to the argument or command following the XECUTE argument upon completion of or exit from the subroutine. The implied routine-body context of this subroutine (necessary for interpretation of a $TEXT or local entry reference in the subroutine) is the routine body containing the XECUTE.

Cross-reference

6.2 paragraph 6 Execution of XECUTE
8.2.13 QUIT

Examples

1.	XECUTE A	If A has the value "S X=1", then X will receive the value 1.
2.	X "S X=B Q:X<3 W Y" W "OUT"	In this example the QUIT in the value of the argument may terminate the execution of the subroutine. The "OUT" is always written.
3.	X "G A" W "OUT"	The "OUT" is never written.
4.	X A,B_" D 3 G "_C,D	XECUTE may have multiple arguments. Note also the use of concatenation for stringing together commands and arguments.
5.	X:Y=1 A_" X B,C",Y	XECUTE with a post-conditional. Note the nesting of XECUTEs.
6.	X @Y,"I @A G "_@D	An example with indirection at two levels.

Z

8.2.20

Syntax

Z [spelling of remainder of command word is specified by the implementor] argument syntax is specified by the implementor

Elaboration

Implementors desiring to offer commands not in the standard are required by the standard to spell the command words beginning with Z.

[Port: Routines designed for portability should not contain Z commands. :Port]

CHAPTER 9

SPECIAL VARIABLES

Table of Contents

CHAPTER 9

SPECIAL VARIABLES

A special variable is an upper-case alphabetic name from a prescribed list, preceded by a $, which, when executed as an expression atom, yields a value which is in System Storage or Partition Storage but is not otherwise explicitly available. The following special variables are defined.

$HOROLOG	The content of the Clock Register.
$IO	The content of the Current Device Designator.
$JOB	The content of the element of the Job Number P-vector corresponding to this partition.
$STORAGE	A measure of space available for the routine and Named Partition Storage.
$TEST	The content of the Test Switch.
$X	The content of CX.
$Y	The content of CY.

$Z[implementor-defined]: as defined by the implementor.

Special variables may be abbreviated to two characters: the $ followed by the first letter of the name.

The syntactic entity special variable is defined as follows.

```
special variable ::= |$H[OROLOG]                |
                     |$I[O]                     |
                     |$J[OB]                    |
                     |$S[TORAGE]                |
                     |$T[EST]                   |
                     |$X                        |
                     |$Y                        |
                     |$Zimplementor-specified   |
```

$HOROLOG 9.1

Name: $HOROLOG

Abbreviation: $H

Value: The content of the Clock Register at the time of execution.

Cross-reference: 4.5 Clock Register

Examples:

1. The following code converts $H to a readable time format such as 11:15:01 AM

```
S TIME=$P($H,",",2),SECONDS=TIME#60
S X=TIME\60,HOURS=X\60,MINUTES=X#60
S SUFFIX=$P("AM,PM",",",HOURS\12+1),HOURS=HOURS#12 S:'HOURS HOURS=12
W HOURS,":",MINUTES,":",SECONDS," ",SUFFIX,!
```

2. The following code converts $H to a date in the form MM/DD/YYYY.

```
S H=$H 21608+$H,LEAPYRS=H\1461,Y=H#1461
S YEAR=LEAPYRS*4+1841+(Y\365),DAY=Y#365
S MO=1 I Y=1460 S DAY=366,YEAR=YEAR-1
F I=31,YEAR#4=0+28,31,30,31,30,31,31,30,31,30,31 Q:DAY'>I  S DAY=DAY-I,MO=MO+1
W MO\10,MO#10,"/",DAY\10,DAY#10,"/",YEAR
```

9.2 **$IO**

Name: $IO

Abbreviation: $I

Value: The content of the Current Device Designator.

Cross-reference: 5.4 Current Device Designator

Examples:

1. `USE 3 S X=$I` — X is given the value 3.

2. `S ^A($I,1)=X` — Scratch files can be set up with data stored by device number.

$JOB

9.3

Name: $JOB

Abbreviation: $J

Value: The content of that element of the Job Number P-vector corresponding to this partition.

Cross-reference: 4.4 Job Number P-vector

Examples:

1. `S ^A($J,2)=Y` — Scratch files can be set up with data stored by job number.

2. `WRITE #,DATE," ",$J` — A way of identifying the source of output.

9.4 $STORAGE

Name: $STORAGE

Abbreviation: $S

Value: The value of $S is not precisely defined by the Standard because it is necessarily somewhat implementation-dependent. $S measures, at the time of its execution, the number of additional characters which may be added to the "partition space" before the system refuses to allow further expansion of the partition space. In the absence of a more precise formula provided by the implementor, the Portability Requirements suggests the following model. Partition space PS is computed as the following sum.

PS = RS + LVS + TRS

RS (routine size) is the number of characters in all lines of the routine body at the top of the Routine Stack (where ls counts as one character and eol counts as two characters).

LVS (local variable storage) is the total number of characters in all Value attributes of all nodes of Named Partition Storage (undefined values are considered empty) plus the total number of data characters (¢ is excluded) in all Name attributes of such nodes plus two times the number of such nodes plus two times the number of pairs of such nodes which are in the immediate ascendant/immediate descendant relationship.

TRS (temporary result storage) is the number of characters of temporary data values, arising from execution of the current line, in existence at the time of execution of $S. These include

1. Characters which have been added to and remain in the line buffer as a result of indirection.
2. Intermediate results in the evaluation of the expression containing the $S due to incomplete parenthesized right-hand operands of binary operators.
3. If the $S occurs in a SET, the data characters in the storage reference node designators which are both to the left of the = and to the left of the $S.

[Port: PS is not to exceed 4000 at any time during execution of a routine designed for portability. :Port]

If TAS is the "total available space" measured in characters at the time of execution of the $S,

$S = TAS - PS .

Note, however, that TAS may be a function of activity outside the partition in which the $S is executed.

Cross-reference: 5.1 Named Partition Storage (See also 4.1)
6.2 paragraph 8 Indirection
Chapter 7 Structure of lines
8.2.15 left-hand side of SET

Examples:

1. I $S<50 G ^OVERFLOW — A test for $S below a threshold permits preparing for a storage overflow.

2. F I=1:1:100 DO 3 I $S<100 D ^FILE

9.5 **$TEST**

Name: $TEST

Abbreviation: $T

Value: The content of the Test Switch (0 or 1). The Test Switch must be defined at the time of execution of $T, ELSE, or argumentless IF.

Cross-reference:
5.3 Test Switch
8.1.4.2 timeouts
8.2.4 ELSE
8.2.9 IF

Examples

1.
```
I X=3 S Y=A
W:$T Y S Y=B
```
This code is equivalent to
```
I X=3 S Y=A
I  W Y
S Y=B
```

2.
```
R X:100 G A:'$T DO B
```
In this example, $T is used to test whether the input was explicitly terminated before the timeout expired. If there was no termination (i.e., if the input string is incomplete), execution continues at A.

3.
```
O 4:10 G A:'$T U 4 W X
```
This is similar to example 3; execution transfers to A if the OPEN is unsuccessful.

$X

9.6

Name: $X

Abbreviation: $X

Value: The content of CX

Cross-reference: 5.5 CX

Examples:

1. `W !,"123" S Y=$X` — Y is given the value 3.

2. `W ! S Y=$X` — Y is given the value 0.

3. `W:$X>72 !` — This code writes a carriage return, line feed if $X indicates that the next output character would be to the right of column 72.

9.7 **$Y**

Name: $Y

Abbreviation: $Y

Value: The content of CY

Cross-reference: 5.5 CY

Examples:

1. `W # S Y=$Y` — Y is given the value 0.

2. `W #!!! S Y=$Y` — Y is given the value 3.

3. `W:$Y>56 #` — If the page is full, skip to a new page.

$Z

9.8

Name: $Zspecified by implementor

Abbreviation: $Zspecified by implementor

Value: Specified by implementor. The Standard requires that all implementor-defined special variables begin with $Z.

[Port: Routines designed for portability should not contain $Z special variables. :Port]

CHAPTER 10

FUNCTIONS

Table of Contents

CHAPTER 10

FUNCTIONS

10.1 General Information

Each MUMPS function has one or more function arguments and, when it is executed as an expression atom, yields a value which is a data string. While the specific syntax of each function is separately prescribed, all functions have the following general syntax.

1. A function is a function name followed by an argument list enclosed in parentheses.
2. A function name is a "$" followed by an upper-case alphabetic name from the prescribed list of function names.
3. In the case of multiple function arguments, adjacent arguments are separated by a comma.

The following function names are defined.

$ASCII selects a character of a string and returns its code as an integer.

$CHAR translates a list of integers into a string of characters whose codes are those integers.

$DATA returns the length of a string, or the number of occurrences of a substring in a string.

$EXTRACT returns a character or substring of a string expression, selected by position number.

$FIND returns an integer specifying the end position of a specified substring within a string.

$JUSTIFY returns the value of an expression, right-justified in spaces within a field of specified size.

$LENGTH returns the length of a string, or the number of occurrences of a substring in a string.

$NEXT identifies the next (in order of defined collating sequence on the last subscript) sibling node of a specified node of Named System or Named Partition Storage.

$ORDER identifies the next (in order of defined collating sequence on the last subscript) sibling node of a specified node of Named System or Named Partition Storage. Both $NEXT and $ORDER use the same collating sequence.

$PIECE returns the string between two specified occurrences of a specified substring within a specified string.

$RANDOM returns a pseudo-random number in a specified interval.

$SELECT returns the value of one of several expressions in a list, selected by the truth values in a second list of expressions.

$TEXT returns the text content of a specified line of the routine body at the top of the Routine Stack.

$VIEW an implementor-defined function available for providing implementation-specific data.

$Z[implementor-defined]
as defined by the implementor.

Function names may be abbreviated to two characters: the $ followed by the first letter of the name.

The syntactic entity function is defined as follows.

```
function ::= | $A[SCII](expression[,integer expression])
             | $C[HAR](integer expression[,integer expression]...)
             | $D[ATA](storage reference)
             | $E[XTRACT](expression[,integer expression][,integer expression])
             | $F[IND](expression,expression[,integer expression])
             | $J[USTIFY](expression,integer expression)
             | $J[USTIFY](numeric expression,integer expression,integer expression)
             | $L[ENGTH](expression[,expression])
             | $N[EXT](storage reference)
             | $O[RDER](storage reference)
             | $P[IECE](expression,expression[,integer expression][,integer expression])
             | $R[ANDOM](integer expression)
             | $S[ELECT](truth-value expression:expression
             |               [,truth-value expression:expression]...)
             | $T[EXT](+integer expression)
             | $T[EXT](line reference)
             | [$V[IEW](argument(s) specified by implementor)]
             | [$Zname specified by implementor(argument(s) specified by
             |               implementor)]
```

10.1.1 Definition of Position Number

If a nonempty string S has n characters, each character is given a unique Position Number in the closed interval [1,n]. The leftmost character of S has the position number 1, the rightmost character of S has the position number n, and intermediate character positions map into the intermediate integers in the expected way.

The reverse mapping $E(S,p) from an integer-valued position number p in S to a string containing zero or one character is defined as follows.

1. If the position number p is less than 1 or greater than n, the value of $E(S,p) is the empty string. (This is also the case for all values of p whenever S is empty.)
2. If the position number p is in the interval [1,n], the value of $E(S,p) is the one-character string whose character is the character of S with position number p.

This reverse mapping is the definition of the two-argument form of $EXTRACT(S,p).

$ASCII

Name: $ASCII

Abbreviation: $A

Syntax: $A[SCII](expression 1 [,integer expression 2])

Value: If integer expression 2 is absent, a default value of 1 is used.

$ASCII yields an integer value n which is the decimal equivalent of the code of the value of $E(expression 1,integer expression 2), such that $A($C(n))=n. If that value is empty, the value of $ASCII is -1.

Cross-reference: 10.1 $E
Appendix A ASCII code table

Examples:

Value of first argument string	Value of function
SET X="ABCDE"	$A(X)=65 $A(X,1)=65 $A(X,2)=66 $A(X,3)=67
SET Y=4	$A(X,Y)=68
SET X="" (the empty string)	$A(X)=-1 $A(X,n)=-1 for all n
SET X="AB"	$A(X,0)=-1 $A(X,3)=-1 $A(X,-7)=-1 $A(X,1.92)=65 (that is, the integer value is used)

10.2.2 **$CHAR**

Name: $CHAR

Abbreviation: $C

Syntax: $C[HAR](integer expression [,integer expression] ...)

Value: $CHAR returns a string whose length is the number of argument expressions which have nonnegative integer values. Each integer expression whose value is in the closed interval [0,127] maps into the ASCII character whose code (expressed as a decimal number) is the integer expression's value; this mapping is order-preserving. Each negative-valued argument maps into no character in the value of $CHAR.

Cross-reference: Appendix A ASCII code table

Examples:

Value of arguments	Value of function
SET X=65,Y=66,Z="GOB"	$C(X)="A" $C(Y)="B" $C(X,Y)="AB" $C(X,Y,67)="ABC" $C(X,-1,Y)="AB" $C(-1)="" (the empty string) $C(0)=the ASCII NUL character $C($A(Z,1),$A(Z,2),$A(Z,3))="GOB"

$DATA

10.2.3

Name: $DATA

Abbreviation: $D

Syntax: $D[ATA](storage reference)

Value: If the node designated by the argument exists, the value of the function is the content of its D attribute. If the node does not exist, the value of the function is 0. If the storage reference is a naked reference, the Naked Indicator must be defined. If the storage reference is a global variable or a naked reference, the Naked Indicator's content may change.

Cross-reference:
4.1 D attribute
5.2 Naked Indicator
12.4 storage reference

Examples:

1. Assume that Named Partition Storage is in its initial state.

Argument values	Function values
Y has no defined value	$DATA(Y)=0
SET Y=100	$DATA(Y)=1
SET Y="AB"	$D(Y)=1
SET A(1)="TEST"	$D(A(1))=1
	$D(A)=10
SET B(1,2)="SAMPLE"	$D(B(1,2))=1
	$D(B(1))=10
SET B(1)="ANOTHER SAMPLE"	$D(B(1))=11
KILL B(1,2)	$D(B(1,2))=0
	$D(B(1))=1 (the KILL may change the D attribute of an ascendant)

2. Assume that the argument is a global variable and that Named System Storage is in its initial state except for the result of executing SET ^A(1,2,3)="TEST".

Action	Value of X	Resulting value of Naked Indicator
SET X=$D(^A)	10	undefined
SET X=$D(^(1))	error	undefined
SET X=$D(^A(1))	10	^A
SET X=$D(^A(99))	0	unchanged
SET X=$D(^(1))	10	unchanged
SET X=$D(^(1,2))	10	^A ¢ 1
SET X=$D(^(2))	10	unchanged
SET X=$D(^(2,3))	1	^A ¢ 1 ¢ 2
SET X=$D(^(3))	1	unchanged
SET X=$D(^(4))	0	unchanged
SET X=$D(^A(1,2))	10	^A ¢ 1
SET ^A(1,2,3,4)="H",X=$D(^A(1,2,3))	11	^A ¢ 1 ¢ 2

$EXTRACT

10.2.4

Name: $EXTRACT

Abbreviation: $E

Syntax:

$E[XTRACT](expression 1[,integer expression 2][,integer expression 3])

Value: If integer expression 2 is absent, a default value of 1 is used in the following as the value of integer expression 2.

If integer expression 3 is absent, the definition given in 10.1.1 is used. If integer expression 3 is present, the value of $EXTRACT is the contiguous substring of expression 1 defined by the following concatenation formula.

If the value of integer expression 3 is not less than the value of integer expression 2,

$E(S,m,n) = $E(S,m)_$E(S,m+1)_ ... _$E(S,n-1)_$E(S,n) .

If the value of integer expression 3 is less than the value of integer expression 2, the value of the function is the empty string.

Cross-reference: 10.1.1 Definition of two-argument form of $E

Examples:

Prior condition	Function Value
SET X="ABCDE"	$E(X)="A" $E(X,1)="A" $E(X,2)="B" $E(X,1,2)="AB" $E(X,1,4)="ABCD" $E(X,0,100)="ABCDE" $E(X,1.9)="A" (the integer value of the argument is used) $E(X,99)="" (the empty string) $E(X,-3)="" (the empty string) $E(X,3,2)="" (the empty string)
SET A(1)="2BC"	$E(A(1),A(1))="B" (the integer interpretation of the second argument is used)

10.2.5

$FIND

Name: $FIND

Abbreviation: $F

Syntax: $F[IND](expression 1,expression 2[,integer expression 3])

Value: If integer expression 3 is absent, or if its value is less than 1, a default value of 1 is used. $FIND searches for an instance of the value of expression 2 as a substring of expression 1. The search is conducted from left to right beginning at the character whose position number is the value of integer expression 3.

If the value of expression 2 is empty, $FIND returns the default or explicit value of expression 3 (see above). If either the value of integer expression 3 is greater than the length of expression 1, or if no instance of expression 2 is found in expression 1, $FIND returns zero. Otherwise, the first (leftmost) instance found is the one reported, as follows. If the position numbers in expression 1 of the found instance of expression 2 are the integers in the interval [i,j], the value of $FIND is j+1.

Cross-reference: 10.1.1 Position Number

Examples:

Prior condition	Function value
SET X="ABCAX"	$F(X,"A")=2
	$F(X,"B")=3
	$F(X,"Z")=0
	$F(X,"ABC")=4
	$F("ABC","ABC")=4
	$F(X,"A",1)=2
	$F(X,"A",2)=5
	$F(X,"A",4)=5
	$F(X,"A",5)=0
	$F(X,"A",100)=0
	$F(X,"")=1
	$F(X,"",4)=4
	$F(X,"",10)=10
SET Y="B"	$F(X,Y)=3
SET Z="1.2W"	$F(X,Y,Z)=3
	$F(X,"",-5)=1

$JUSTIFY

10.2.6

Name: $JUSTIFY

Abbreviation: $J

Syntax:

$J[USTIFY](expression 1,integer expression 2 | numeric expression 1,integer expression 2,integer expression 3)

Value: The two-argument form right-justifies expression 1 in a field of integer expression 2 spaces. Let L1 be the length of expression 1 and let N2 be the value of integer expression 2. If N2 ≤ L1 there is no truncation; the value of $J is the value of expression 1. If N2 > L1 the value of $J is N2-L1 spaces on the left concatenated with the value of expression 1 on the right.

In the three-argument form, also let N3 be the value of integer expression 3. This form edits the value of numeric expression 1, right-justified, in a field of N2 spaces with N3 decimal places. Specifically, let R be the value of numeric expression 1 after rounding to N3 fraction digits, including possible trailing zeros. (If N3=0, R contains no decimal point; if the value of R is between -1 and 1, and N3>0, R does have a zero to the left of the decimal point.) The value returned by $J is $J(R,N2).

[Port: Negative values of N3 are reserved for future expansion of the definition of $J, and therefore should be avoided by implementors as well as users. :Port]

Cross-reference:

Examples:

Prior condition	Function value
SET X=12.35	$J(X,6)=" 12.35" $J(X,5)="12.35" $J(X,4)="12.35" $J(X,3)="12.35" $J(X,7,4)="12.3500" $J(X,7,3)=" 12.350" $J(X,7,2)=" 12.35" $J(X,7,1)=" 12.4" $J(X,7,0)=" 12"
SET Y=197	$J(Y,7,2)=" 197.00"
SET Z=5.4	$J(Z,7,2)=" 5.40"
SET W=1.487	$J(W,7,2)=" 1.49" $J(W,7,1)=" 1.5" $J(W,7,0)=" 1"
	$J(X/Y,5,3)=" 0.063"
	$J(-W/Z,9,4)=" -0.2754"

$LENGTH

10.2.7

Name: $LENGTH

Abbreviation: $L

Syntax: $L[ENGTH](expression[,expression])

Value: There are two forms of $LENGTH.

$LENGTH(expression) returns the integer number of characters in the value of expression. If the value of expression is empty, $L returns zero.

$LENGTH(expression 1,expression 2) returns the integer number plus one of the nonoverlapping occurrences of expression 2 in expression 1. If the value of expression 2 is empty, this form of $L returns zero.

Cross-reference:

Examples:

Prior condition	Function value
SET X="ABC"	$L(X)=3
SET X="123456789"	$L(X)=9
SET X="" (the empty string)	$L(X)=0
SET X="ABCDBCABCABCD"	$L(X,"AB")=4
	$L(X,"DC")=1
	$L(X,"ABCD")=3
	$L(X,"")=0

10.2.8 **$NEXT**

Name: $NEXT

Abbreviation: $N

Syntax: $N[EXT](storage reference)

Value: $NEXT is included for backward compatibility. The use of the $ORDER function (10.2.8.5) is strongly encouraged in place of $NEXT, as the two functions perform the same operation except for the different starting and ending condition of $NEXT.

Only subscripted (level 2 or greater) forms of the argument are permitted. In addition to the normal subscript values, the value of the rightmost subscript may also be -1. If the argument is a naked reference, the Naked Indicator must be defined.

$NEXT returns a value which is a subscript according to a subscript ordering sequence. This ordering sequence is specified with the aid of the definitional function, CO, which is defined in $ORDER (10.2.8.5).

Let the argument be of the form

N(s1,s2, ..., sn-1,sn)

where sn is constrained as stated. $NEXT reports the existence and identity of a storage node denoted by the storage reference

N(s1,s2, ..., sn-1, t)

such that the node has a nonzero D attribute, and, if sn is not -1, such that

1. CO(sn,t)=t; and
2. CO(s,t)=s for all s not equal to t such that CO(sn,s)=s.

If sn is -1, then CO(s,t)=s for all s not equal to t such that CO("",s)=s.

If such a node exists, the value of $NEXT is t; if no such node exists, the value of $NEXT is -1.

Note that $NEXT will return ambiguous results for arrays which have negative numeric subscript values.

Cross-reference:
4.1 Levels of storage
4.1 D attribute
12.4 storage reference
10.2.8.5 $ORDER function

Examples: Assume that Named Partition and System Storage are in their initial states except for the results of execution of SET ^A(1,2,3)="TEST",^A(4)="SAMPLE".

Action	Value of X	Resulting value of Naked Indicator
SET X=$N(^A)	error	undefined
SET X=$N(^A(-1))	1	^A
SET X=$N(X(4))	-1	unchanged
SET X=$N(^(1))	4	unchanged
SET X=$N(^(4))	-1	unchanged
SET X=$N(^(4,3))	-1	undefined
SET X=$N(^A(1,-1))	2	^A ç 1
SET X=$N(^(2,-1))	3	^A ç 1 ç 2
SET X=^(3)	"TEST"	unchanged
SET X=$N(^(3))	-1	unchanged
SET X=$N(^B(1,2))	-1	^B ç 1

10.2.8.5 **$ORDER**

Name: $ORDER

Abbreviation: $O

Syntax: $O[RDER](storage reference)

Value: Only subscripted (level 2 or greater) forms of the argument are permitted. In addition to the normal subscript values, the value of the rightmost subscript may also be empty. If the argument is a naked reference, the Naked Indicator must be defined.

$ORDER returns a value which is a subscript according to a subscript ordering sequence. This ordering sequence is specified with the aid of a definitional function, CO, as follows:

> For strings s and t, CO(s,t) returns t when t follows s in the ordering sequence; otherwise, CO(s,t) returns s.

Let m and n be strings satisfying the definition of numeric data values (11.1.2), and u and v be nonempty strings which do not satisfy this definition. The following cases define the ordering sequence:

a. CO("",s) = s.
b. CO(m,n) = n if n > m; otherwise, CO(m,n) = m.
c. CO(m,u) = u.
d. CO(u,v) = v if v]u; otherwise, CO(u,v) = u.

In other words, all strings follow the empty string, numerics collate in numeric order, numerics precede nonnumeric strings, and nonnumeric strings are ordered by the conventional ASCII collating sequence.

Let the argument be of the form

N(s1,s2, ..., sn-1,sn)

where sn is constrained as stated. $ORDER reports the existence and identity of a storage node denoted by the storage reference

N(s1,s2, ..., sn-1,t)

such that

1. the node has a nonzero D attribute;

2. CO(sn,t)=t; and
3. CO(s,t)=s for all s not equal to t such that CO(sn,s)=s.

If such a node exists, the value of $ORDER is t; if no such node exists, the value of $ORDER is the empty string.

Cross-reference: 4.1 Levels of storage
4.1 D attribute
12.4 storage reference

Examples: Assume that Named Partition and System Storage are in their initial states except for the results of the following executions:

```
SET LN="DOE",FN="JOHN",^A(LN,FN,-1.2)=3,^(0)=25669
SET ^(1.34)=17,^("3/47")=2,^("age")=56,^("Sex")="M",^("TYPE")="0-"
```

Action	Value of X	Resulting Value of Naked Indicator
SET X=$O(^A)	error	undefined
SET X=$O(^A(""))	"DOE"	^A
SET X=$O(^A("DOE"))	""	unchanged
SET X=$O(^A("DOE",""))	"JOHN"	^A ¢ DOE
SET X=$O(^("JOHN",""))	-1.2	^A ¢ DOE ¢ JOHN
SET X=$O(^(-1.2))	0	unchanged
SET X=$O(^(0))	1.34	unchanged
SET X=$O(^(1.34))	"3/47"	unchanged
SET X=$O(^("3/47"))	"Sex"	unchanged
SET X=$O(^("Sex"))	"TYPE"	unchanged
SET X=$O(^("TYPE"))	"age"	unchanged
SET X=$O(^("age"))	""	unchanged
SET X=$O(^B(1,2))	""	^B ¢ 1
SET X=$O(^(""))	""	unchanged

10.2.9 **$PIECE**

Name: $PIECE

Abbreviation: $P

$P[IECE](expression 1,expression 2[,integer expression 3 [,integer expression 4]])

Value: If integer expression 3 is absent, a default value of 1 is used in the following as the value of integer expression 3.
If integer expression 4 is absent, a default value which is the value of integer expression 3 is used.
Let S1 denote the value of expression 1, let S2 denote the value of expression 2, let N3 denote the value of integer expression 3, and let N4 denote the value of integer expression 4.

$PIECE(S1,S2,N3,N4) searches for, and returns as its value, a substring of S1 whose left and right boundaries are delimited by values specified by S2, N3, and N4. Typically, the substring is bounded on the left by (but does not include) the N3-1th occurrence of S2 in S1, counting from the left end of S1, and the substring is bounded on the right by (but does not include) the N4th occurrence of S2 in S1, counting from the left end of S1. Since empty substrings may be found anywhere in any number required, $P returns an empty string if S2 is empty.

Any of the following conditions causes $PIECE to return the empty string.

1. N3 > N4
2. N4 < 1
3. There are fewer than N3-1 distinct nonoverlapping instances of S2 in S1.
4. S2 is empty.

Provided that none of the above conditions holds, the value of $PIECE is the contiguous substring of S1 which is both to the right of the N3-1th occurrence of S2 in S1 and to the left of the N4th occurrence of S2 in S1.

The occurrences of S2 in S1 are counted from the left end of S1 and are considered to be nonoverlapping. Thus, there are two occurrences of "ABAB" in "ABABABABAB", in the sense of the terms used here.

If N3 is 1 or less, the selected substring includes the left end of S1. If N4 is greater than the number of occurrences of S2 in S1, the selected substring includes the right end of S1.

Cross-reference:

Examples:

Prior condition	Function value
SET X="ABC*DEF"	$P(X,"*")="ABC" $P(X,"*",1)="ABC" $P(X,"*",2)="DEF" $P(X,"*",3)="" (the empty string) $P(X,"B",1)="A"
SET Y="B"	$P(X,Y,1)="A" $P(X,Y,2)="C*DEF" $P(X,"/",1)="ABC*DEF"
SET Z="A.B.C.D"	$P(Z,".",1)="A" $P(Z,".",2,3)="B.C" $P(Z,".",1,100)="A.B.C.D" $P(Z,".",3,2)="" (the empty string) $P(Z,"",1,100)=""

10.2.10 **$RANDOM**

Name: $RANDOM

Abbreviation: $R

Syntax: $R[ANDOM](integer expression)

Value: Let N denote the value of the argument. N must be positive. $R(N) returns a random or pseudo-random integer uniformly distributed in the closed interval [0,N-1].

Cross-reference:

Examples:

1.	SET X=$RANDOM(N)	X will be given an integer value between 0 and N-1.
2.	SET Y=$R(2)	Y will randomly be given either the value 0 or 1.
3.	SET Y=$R(1)	Y is 0.

$SELECT

10.2.11

Name: $SELECT

Abbreviation: $S

Syntax:

$S[ELECT](truth-value expression:expression

[,truth-value expression:expression]...)

Value: Each argument of $SELECT is an ordered pair separated by a colon: truth-value expression:expression . The function may contain any positive number of arguments. At least one truth-value expression must have a value 1 (true).

Without evaluating the corresponding expression, each truth-value expression is evaluated, one at a time, in left-to-right order. This evaluation stops at the first argument for which the truth-value expression has the value 1. In this argument, and in no other, the expression is evaluated and this latter value becomes the value of $SELECT. Notice that no other expression need have a defined value.

Cross-reference:

Examples:

Prior condition	Function value
SET X=1	$S(X=1:8,2=3:0)=8 $S(X=1:8,2=2:0)=8 $S(X=2:8,2=2:0)=0 $S(X=2:8,2=3:0)=error; one argument must have a true truth value.
SET Y="B"	$S(X=3:8,Y="B":"HELLO",X=1:13)="HELLO" $S(X=Y:B(1),Y="A":^(2,3),1:3)=3 Note that this function call does not use or alter the Naked Indicator.
SET ^A(1)="TEST"	$S(X=1:^A(1),Y:S)="TEST" The Naked Indicator is changed to ^A .

10.2.12 **$TEXT**

Name: $TEXT

Abbreviation: $T

Syntax: $T[EXT](| + integer expression / line reference |)

Value: The argument denotes either a routine name or a line of the routine which is at the top of the Partition Stack. This denotation is made in either of the following ways.

1. If the argument has the form + integer expression, its value, which we shall denote by N, must be nonnegative. If N=0, the routine name of the routine is denoted, and is returned by $TEXT. If N>0, the line denoted is the Nth line of the routine body. (The first line is associated with N=1.)
2. If the argument has the form line reference, the line is that denoted by the line reference, according to the method given in Chapter 7.

If the argument of $TEXT, by virtue of a line reference containing a label not appearing in a line head of the routine body or by virtue of a too-large value of an integer expression, does not denote a line, or if there is no routine at the top of the Partition Stack, the value of $TEXT is the empty string. Otherwise, the value of $TEXT is the value of the line denoted, except for the following changes: ls is replaced by one space (SP), and eol is deleted.

Cross-reference: Chapter 6 Partition Stack
Chapter 7 routine body
Chapter 7 line reference

Examples:

Assume that the following routine body is at the top of the Partition Stack.

```
ABC       ;SAMPLE
          SET X=3 WRITE Y
          READ X
Z         WRITE !,X
```

Then:

```
$T(0)="ABC"  (given that the routine name of this routine body is ABC)
$T(ABC)="ABC ;SAMPLE"
$T(Z)="Z WRITE !,X"
$T(ABC+1)=" SET X=3 WRITE Y"
$T(ABC+2)=" READ X"
$T(+1)="ABC ;SAMPLE"
$T(+3)=" READ X"
$T(Z+1)=""
$T(@A+2)=" READ X" (given that A="ABC")
$T(ABC+K)=" READ X"  (given that K=2)
$T(+B)=" READ X"  (given that B=3)
```

The following two lines write out the routine body containing them.

```
F I=1:1 S X=$T(+I) Q:X=""  W X,!
W #
```

10.2.13 **$VIEW**

Name: $VIEW

Abbreviation: $V

Syntax: $V[IEW](argument syntax specified by implementor)

Value: $VIEW is an implementation-specific function available to the implementor who may wish to make available to programs certain data not otherwise available. Each implementation must recognize and accept the $VIEW function, regardless of any interpretation given to it.

[Port: Routines designed for portability should not contain calls of the $VIEW function. :Port]

$Z

10.2.14

Name: $Zremainder of name specified by implementor

Abbreviation: $Zremainder of abbreviation specified by implementor

Syntax:

$Zspecified by implementor(argument syntax specified by implementor)

Value: Implementors desiring to offer functions not in the standard are required by the standard to spell their function names beginning with $Z.

[Port: Routines designed for portability should not contain calls of any $Z function. :Port]

CHAPTER 11

EXPRESSIONS

Table of Contents

CHAPTER 11

EXPRESSIONS

11.1 General Rules Governing Expressions

11.1.1 Syntax of expression

An expression is a substring of a command which, when executed, yields a value. The execution of expressions is the principal means by which values are obtained for subscripts, for arguments of functions, and for assignment to storage nodes.

Expressions may be arbitrarily complex, subject to certain practical limits such as that placed on line length for the sake of portability (see Chapter 7); however, there are typically just two atomic elements of which expressions are built: expression atoms and binary operators. An expression atom (discussed in Chapter 12) is the simplest form of an expression; it is the atomic value-yielding substring of a line of which more complex expressions are compounded. Binary operators provide the linkages between expression atoms to form the more complex expressions.

As is almost every other process in the execution of a MUMPS routine, the execution of an expression is a progressive process which accompanies a left-to-right scan of characters of the line. This basic progressive structure is revealed in the syntax definition of expression.

$$\underline{\text{expression}} \quad ::= \quad \left| \begin{array}{l} \underline{\text{expression atom}} \\ \\ \underline{\text{expression}}\ \underline{\text{expression tail}} \end{array} \right|$$

The right-hand side of the above recursive formulation is syntactically equivalent to the following simpler, iterative, formulation

$$\underline{\text{expression atom}}\ [\underline{\text{expression tail}}]\ \ldots$$

but the recursive formulation is chosen to correlate more closely with the calculation model which is presented. Every expression tail is in two parts: a binary operator on the left and an expression atom or pattern on the right.

11.1.2 MUMPS Values

The universe V within which all MUMPS values fall is the set of all strings of characters chosen from the 7-bit ASCII set. (See Appendix A for a list of these 128 ASCII characters.) This universe includes the empty string. Except for certain practical limits such as storage capacity, given a value chosen from this universe, a MUMPS routine can be written to produce that value.

[Port: The practical universe in which all values produced by MUMPS routines designed for portability must fall contains all ASCII strings whose lengths are 255 or less and no strings whose lengths exceed 255. :Port]

There are four important subsets of V: the subset Ve of numeric values expressed in exponential notation, its subset Vn of numeric values without exponents, its subset Vi of integer values, and its subset Vt of truth values. These subsets form a descending nested sequence in the order given. The subsets are defined by the syntax of the string which are their elements. This relationship between set membership and syntax is expressed in the following table.

Strings in the value subset	All have the syntax of	Defined in
Ve	exponential value	11.1.3
Vn	numeric value	11.1.2
Vi	integer value	11.1.2
Vt	truth value	11.1.2

The syntax definitions of numeric value, integer value, and truth value are given below.

truth value ::= | 0 | 1 |

nonzero digit ::= | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

digit ::= | 0 | nonzero digit |

integer value ::= | 0 | [-] nonzero digit [digit] ... |

[Port: The number of digits of any integer value produced by the execution of a MUMPS routine designed for portability may not exceed nine. :Port]

fraction value ::= . [digit] ... nonzero digit

(Note that the single period denotes ".", whereas the group of three periods denotes optional repetition of the [digit] syntactic element.)

numeric value ::= | 0 | [-] nonzero digit [digit] ... [fraction value] | [-] fraction value |

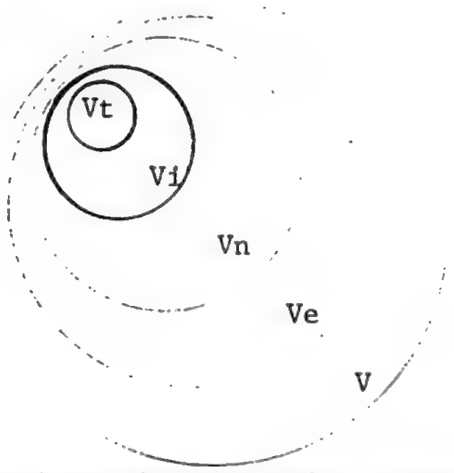
Each integer value or numeric value uniquely denotes a real number; the number denoted is the obvious one. The truth value 0 denotes False; the truth value 1 denotes True.

[Port: A MUMPS routine designed for portability may produce only those numeric values denoting zero and the numbers whose absolute values are in the closed interval

$$[10^{-25}, 10^{25}] \quad .$$:Port]

11.1.3 Interpretation Operations

The value sets V, Ve, Vn, Vi, and Vt form a nested sequence of sets, as depicted in the following Venn diagram.



That is, every truth value is an integer value, every integer value is a numeric value, every numeric value is an exponential value, and every exponential value is a MUMPS value. Many operators, some examples of which are listed here, have natural definitions, however, only on one of these subsets of V but not on the whole of V.

1. Arithmetic is defined on numeric values.
2. The Boolean operators are defined on truth values.
3. Certain functions, such as $CHAR and EXTRACT, have some arguments which are assumed to be integer values.

It is a principle of design of the MUMPS language that any operation which is defined on a subset of V will not fail to operate if an actual argument falls outside that subset. Rather, any operation which requires such a restricted argument automatically performs an "interpretation" operation on each such argument before the operation itself is performed. The appropriate interpretation operation maps all MUMPS values into the required subset (while not changing those values already having the correct syntax), thereby guaranteeing that all argument values will meet the restrictions assumed in the definition of the operation.

In each argument of a function, operator, command, etc., the requirement for a particular interpretation operation is denoted by the syntactic form of the argument.

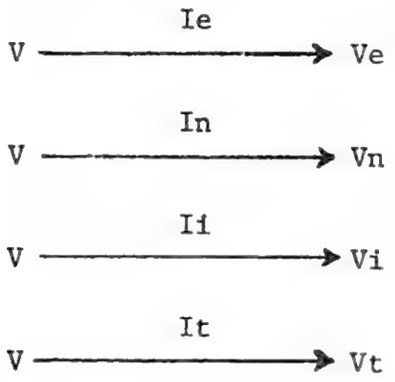
Syntactic form of argument	Required subset of values	Interpretation operation performed on argument prior to execution
expression, expression atom	V	none
exponential expression, exponential expression atom	Ve	exponential interpretation Ie
numeric expression, numeric expression atom	Vn	numeric interpretation In
integer expression, integer expression atom	Vi	integer interpretation Ii
truth-value expression, truth-value expression atom	Vt	truth-value interpretation It

Note that in the routine itself, these specialized argument forms have the same syntax as their more general forms.

exponential expression atom ::= expression atom
numeric expression atom ::= expression atom
integer expression atom ::= expression atom
truth-value expression atom ::= expression atom

exponential expression ::= expression
numeric expression ::= expression
integer expression ::= expression
truth-value expression ::= expression

Each interpretation operation is a mapping from the whole of V onto the required subset of values, which does not alter those values already in the required subset. This is expressed schematically as follows.



Whereas MUMPS arithmetic operators do not produce results in exponential or "scientific" form, the numeric interpretation In correctly interprets argument data in this form and converts it to numeric form. This is the reason for the introduction of the set Ve, even though exponential values per se are not employed in the definition of the arithmetic operations. (In Chapter 12, the exponential form of data is used more explicitly in connection with the interpretation of the numeric literal form of expression atom.) The syntax definition of exponential value, which defines the elements of Ve, follows.

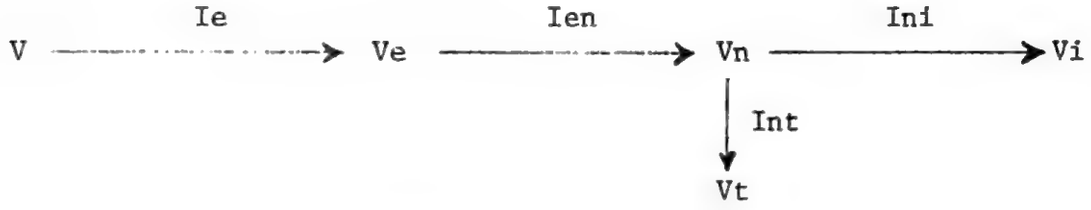
digit sequence ::= digit [digit] ...

mantissa ::= | digit sequence [. digit sequence] | . digit sequence |

exponent ::= E [+ -] digit sequence

exponential value ::= [-] mantissa [exponent]

The interpretations In, Ii, and It then break down into sequences of more than one of the primitive interpretations Ie, Ien, Ini, and Int, as expressed in the following diagram and table.



To obtain the interpretation	Apply the sequence
In	Ie followed by Ien
Ii	In followed by Ini
It	In followed by Int

11.1.3.1 The Interpretation Ie

The transformation of any data string S into one in Ve proceeds in two steps. The first simplifies any leading signs; the second takes the largest head satisfying exponential value.

> Definition of Head. The head of a string with respect to a given point which is immediately to the right or left of any character of the string is defined to be that substring which contains all characters to the left of the point and no characters to the right of the point. Note that a head may be empty, or it may be the entire string.

1. Apply the following initial-sign reduction rules to S as many times as possible, in any order.

   a. If S is of the form + T, then remove the +. (Shorthand: + T → T)
   b. - + T → - T
   c. - - T → T

2. Apply whichever of the two following cases is appropriate.

   a. If the leftmost character of S is not "-", the result is the longest head of S having the syntax of exponential value.

   b. If S is of the form - T, take the longest head of S which has the syntax of exponential value. If the mantissa part of the result denotes 0 (or -0), make the whole value "0".

11.1.3.2 The Interpretation Ien

1. If the mantissa part of the argument has no ".", place one at the right end of the mantissa.

2. The presence of an exponent is recognized by the "E". If it is absent, skip step 3.

3. If the exponent has a + or has no sign, move the "." a number of decimal digit positions to the right in the mantissa equal to the number denoted by the exponent, appending zeros to the right of the mantissa as necessary. If the exponent has a minus sign, move the "." a number of decimal digit positions to the left in the mantissa equal to the absolute value of the number denoted by the exponent, appending zeros to the left of the mantissa as necessary.

4. Delete the exponent and any leading or trailing zeros of the mantissa.

5. If the rightmost character is ".", remove it.

6. If the result is empty or "-0", make it "0".

11.1.3.3 The Interpretation Ini

1. The presence of a fraction value is recognized by the ".". If it is present, delete the fraction value.

2. If the result is empty or "-", make it "0".

11.1.3.4 The Interpretation Int

If the number is not "0", make it "1".

11.1.4 Calculation of expression Values

The syntax of an instance of expression has one of the two forms

expression atom

or

expression expression tail .

Chapter 12 prescribes the value of each expression atom. If the expression has the form expression atom, then the value of the expression is the value of the expression atom.

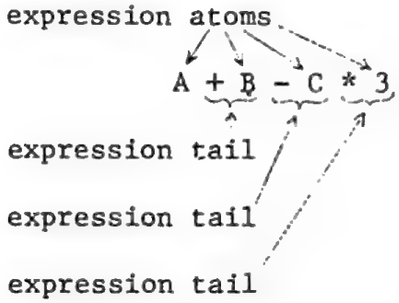
Otherwise, the expression is of the form

expression expression tail ,

and the following general comments apply to its evaluation.

The MUMPS scan of an expression is left to right. At the time that the expression is to be evaluated, the rightmost character of the expression tail has just been scanned. Therefore, the values of all component expressions and/or expression atoms are already known. Specifically, the value of the left-hand expression is known, and the value of any expression atom in the expression tail is known.

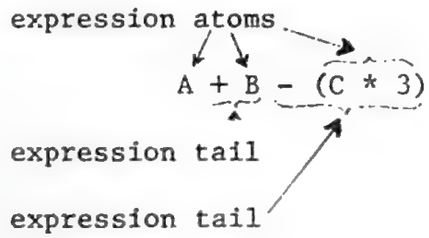
Because an evaluation occurs immediately after each expression tail is scanned, and because the direction of scan is left-to-right, an expression containing several expression tails, e.g.,



is naturally grouped from the left. That is, any implied parentheses are as follows.

(((A + B) - C) * 3)

If it is desired to force a different order of calculation, parentheses may be used to insure that certain combinations of symbols be treated as expression atoms, to wit:



11.2 Types of Expression Tails

It may be predicted from examining just the binary operator of the following expression tail

expression expression tail

in what value class, Vt, Vi, Vn, or V, the value of the overall expression will fall. This fact is the basis for grouping the expression tails into the following four groups, each of which "restricts" its resulting value into the named value class.

1. Nonrestricting expression tails. The only operator in this group is string concatenation (_).
2. Numeric-value restricting expression tails. This group contains the arithmetic operators + - / * #.
3. Integer-value restricting expression tails. The only operator in this group is the integer-quotient divide (\ ).
4. Truth-value restricting expression tails. This group contains the relational operators (= < >] [ ), the logical operators (& !), and the pattern-match operator (?). Any of these operators may be immediately prefixed with the not-prefix (') which reverses the truth value of the result.

The syntax definition of expression tail follows.

```
expression tail  ::=  | | string operator          |  expression atom |
                      | | arithmetic operator      |                  |
                      | | ['] truth operator       |                  |
                      |                                                 |
                      |          ['] ? pattern                          |

string operator  ::=  _

arithmetic operator  ::=  | + |
                          | - |
                          | * |
                          | / |
                          | \ |
                          | # |

truth operator  ::=  | relation |
                     |          |
                     | logical  |

relation  ::=  | < |
               | > |
               | = |
               | [ |
               | ] |

logical  ::=  | & |
              |   |
              | ! |
```

11.2.1 Nonrestricting Expression Tails

Operators: _ (string concatenation)

Expression syntax: expression 1 _ expression atom 2

Value calculation: The value of the resulting expression is the result of concatenating the value of expression 1 with the value of expression atom 2, with expression 1 on the left.

Cross-reference:

Examples: Let X have the value "DE".

Expression	Value
"A"_"B"	AB
"A"_"B"_"C"	ABC
"A"_"1"_23	A123
"AB"_X_(3*4)	ABDE12
1_2	12
1_2*3	36
1_(2*3)	16

11.2.2 Numeric-value Restricting Expression Tails

Operators:

+ (algebraic addition)
- (algebraic subtraction)
* (algebraic multiplication)
/ (algebraic division)
(modulo)

Expression syntax:

numeric expression 1 | + - * / # | numeric expression atom 2

Value calculation: [Port: Routines designed for portability should not contain calculations requiring more than nine significant digits of accuracy in any intermediate or final values. :Port]

The numeric interpretation In is taken of both arguments.

The value of the result is defined by the real number which it denotes, as follows.

+ produces the algebraic sum.

- produces the algebraic difference.

* produces the algebraic product. [Port: The value of the result may not be uniquely defined if the number of significant digits in the product exceeds nine. :Port]

/ produces the algebraic quotient. Note that the sign of the quotient is negative if and only if one argument is positive and one argument is negative. Division by zero is not permitted. [Port: The value of the result may not be uniquely defined if the number of significant digits of the quotient exceeds nine or if the quotient contains a nonterminating fraction. :Port]

produces the value of the left argument modulo the right argument. # is defined only for nonzero values of its right argument, as follows.
A # B = A - (B * floor(A/B))
where floor(x) = the largest integer $\leq$ x.
[Port: The value of the result may not be uniquely defined if the number of significant digits in the result exceeds nine. :Port]

Cross-reference: 11.1.3 Numeric interpretation In

Examples: Let A=0,B=1,C=2.

Expression	Value
2+3	5
2*3+4	10
4+2*3	18
(4+2)*3	18
4+(2*3)	10
4+(2/3)	4.66666667 (approximately)
1+.123456789	1.12345679 (approximately)
A-B-C	-3
A-(B-C)	1
B#C	1
C#B	0
B+2*C	6
B+(2*C)	5

11.2.3 Integer-value Restricting Expression Tails

Operator: \ (integer-quotient divide)

Expression syntax: numeric expression 1 \ numeric expression atom 2

Value calculation: The value of the result is the integer interpretation Ii of the value of

numeric expression 1 / numeric expression atom 2 .

[Port: The value of the result may not be uniquely defined if the number of significant digits in the result exceeds nine. :Port]

Cross-reference: 11.2.2 division operator /

Examples:

Expression	Value
4+2\3	2
4+(2\3)	4

11.2.4 Truth-value Restricting Expression Tails

Operators:

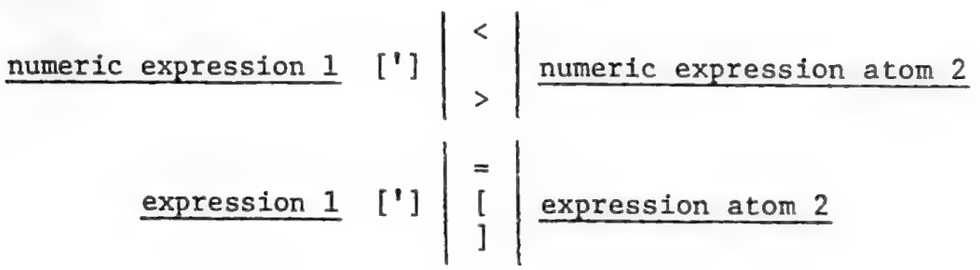
Relational operators (see 11.2.4.1)
- = (string identity)
- < (algebraic less than)
- > (algebraic greater than)
- [(string contains)
-] (string follows)

Logical operators (see 11.2.4.2)
- & (and)
- ! (or)

Pattern-match operator (see 11.2.4.3)
- ?

11.2.4.1 Relational Operators

Expression syntax:



Value calculation: In the case of all of these operators, denoted here by op,

A 'op B has the same value as '(A op B) .

A op B has the value 1 if the relation it expresses is true; it has the value 0 otherwise.

The inequalities > and < compare the numeric interpretations of their arguments; the relation which they express is the conventional algebraic "greater than" or "less than".

The operator = yields the value 1 if and only if the two arguments (without taking any interpretation) are identical strings. If both arguments are known to be in numeric-value form, the uniqueness of the numeric representation permits = to be used to test for numeric equality. [Port: The effects of inexact arithmetic must be taken into account if any truncation has taken place in the calculation of an argument. :Port]

The relation [is called "contains". A [B is true if and only if B is a contiguous substring of A; that is, A [B has the same value as ''$F(A,B). The empty string is a substring of every string.

The relation] is called "follows". A] B is true if and only if A follows B lexicographically in the conventional ASCII collating sequence. A is defined to follow B if and only if any of the following is true.

a. B is empty and A is not.
b. Neither A nor B is empty, and the leftmost character of A has a numerically greater ASCII code than the leftmost character of B.
c. There exists a positive integer n such that A and B have identical heads of length n and the remainder of A follows the remainder of B.

11.2.4.2 Logical Operators

Expression syntax:

truth-value expression 1 ['] | & / ! | truth-value expression atom 2

Value calculation: & and ! are given the names "and" and "or", respectively.

$$A\ !\ B = \begin{cases} 0 \text{ if both A and B have the value 0} \\ 1 \text{ otherwise} \end{cases}$$

$$A\ \&\ B = \begin{cases} 1 \text{ if both A and B have the value 1} \\ 0 \text{ otherwise} \end{cases}$$

The dual operators '& and '! are defined by:

A '& B = '(A & B)
A '! B = '(A ! B) .

11.2.4.3 Pattern-match operator

Expression syntax:

expression 1 ['] ? pattern

```
                    | pattern atom [pattern atom] ...                  |
        pattern ::= |                                                  |
                    | @ expression atom V pattern                      |

                    | integer literal |  | string literal |
   pattern atom ::= |                 |  |                |
                    |        .        |  | pattern code   |

                    | C |
                    | N |
                    | P |
   pattern code ::= | A | ...
                    | L |
                    | U |
                    | E |
```

Value calculation: S '? P is defined to have the same value as '(S ? P).

A pattern is a concatenated list of one or more pattern atoms. Let n be the number of pattern atoms in pattern. S ? pattern is true if and only if there exists a partition of S into n substrings

S = S1 S2 ... Sn

Such that there is a one-to-one order-preserving correspondence between the Si and the pattern atoms, and each Si "satisfies" its respective pattern atom. Note that some of the Si may be empty.

Each pattern atom consists of two parts: a pattern code or a string literal, which is used as the basis for a comparison with Si, and a preceding multiplier: either a numeric literal specifying an exact number of iterations or the "indefinite multiplier" "." .

If the integer literal multiplier is present, Si satisfies the pattern atom if and only if Si is a concatenation of integer literal substrings, each of which satisfies the string literal or pattern code of the pattern atom.

If the indefinite multiplied "." is present, Si satisfies the pattern atom if it is a concatenation of any number (including zero) of substrings, each of which satisfies the string literal or pattern code of the pattern atom.

A substring satisfies a string literal if and only if it is identical to the value of the string literal.

A pattern code is a string of letters, each of which represents a class of characters, as follows.

Letter	Class of characters
C	33 Control characters, including DEL
N	10 Numeric characters
P	33 Punctuation characters, including SP
A	52 Alphabetic characters
L	26 Lower-case alphabetic characters
U	26 Upper-case alphabetic characters
E	Everything (the Entire set of characters)

The class of characters represented by a pattern code is the union of the classes of characters represented by each of the letters of the pattern code. A substring satisfies a pattern code if and only if each of the substring's characters falls into the class of characters represented by the pattern code.

11.2.4.4 Examples of Truth-valued Operators

Relational and logical operators

Let X=3,Y="A"

Expression	Value
1=1	1
1=0	0
(X=3)!(1=1)	1
(X=3)+1	2
X=3+1	2
X>Y	1
X>Y=1	1
Y'<X=1	0
X=3!(Y="A")	1
Y="A"_"B"	1B
"MGH"]Y<3	1

Pattern-match operator

Prior condition	Expression	Value
SET X="123456"	X?.N	1
	X?2N4E	1
	X?1"12"4N	1
	X?.A	0
	X?.N1"6"	1
	X?.E1"3".N	1
SET X="A12B12C"	X?.N	0
	X?.NA	1
	X?2NA.E1"1".E	1
SET X="TESTTESTtest"	X?2"TEST".E	1
	X?1"TEST".E	1
	X?8U4L	1
	X?8A1"test"	1
	X?.U1"test"	1
	X?.A2L	1
SET X="12.34 dollars"	X?.N1P.E	1
	X?2N1P2N1" ".L	1
SET X=12.34	X?.N	0
SET X=1234.	X?.N	1
SET X="1234."	X?.N	0
SET X=$C(7)_$C(10)	X?2C	1
	X?.E1C.E	1

CHAPTER 12

EXPRESSION ATOMS

Table of Contents

CHAPTER 12

EXPRESSION ATOMS

12.1 Syntax of expression atom

An expression atom is the atomic value-yielding component of an expression. (See 11.1.1 for a discussion of the role of expression atoms in expressions.) When executed, an expression atom yields a value. This chapter describes the syntax and value of each form of expression atom.

The syntax definition of expression atom follows. To the right, there is a cross-reference to the detailed discussion of each form of expression atom.

			Defined in
expression atom	::=	(expression)	12.1.1
		special variable	Chapter 9
		function	Chapter 10
		string literal	12.2
		numeric literal	12.3
		storage reference	12.4
		' truth-value expression atom	12.5
		|+| |-| numeric expression atom	12.5

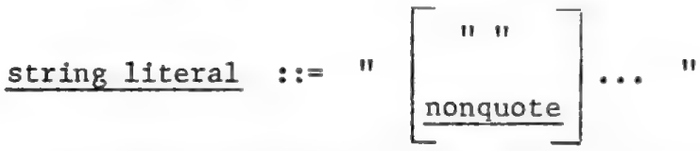
12.1.1 Use of Parentheses to Form Expression Atoms

Any calculation procedure which may be expressed as an expression may be used to compute the value of an expression atom by writing the expression atom as (expression) . Since the value of an expression's component expression atoms must be computed prior to the computation of the value of the expression, this use of parentheses is the basis for controlling the order of evaluation of the components of an expression. (Section 11.1.4 contains more discussion and an example on this subject.)

12.2 String Literals

A "literal" is an expression atom whose value is a function only of its spelling. There are two types of literals, string literals (discussed here) and numeric literals (discussed in 12.3).

The presence of a string literal is indicated by the quotes which are its first and last characters. To define the syntax of string literal one needs to define the class of characters nonquote. In the interest of space, this definition is an informal one. A nonquote is any single ASCII graphic except for the quote sign (").



In words, a string literal is bounded by quotes (considered to be part of the literal), and it contains within the quotes any string of graphics, except that when quotes occur inside, they occur in adjacent pairs. The value of an instance of string literal is the same as its spelling except that the following deletions are made.

1. Neither of the two bounding quotes is in the value.
2. Whenever a quote pair occurs inside the bounding quotes, one quote of the pair is deleted.

The string literal denoting the empty string is written "" .

12.3 Numeric Literals

The presence of a numeric literal (in a context in which an expression atom is appropriate) is indicated by its first character, which must be either a digit or a decimal point. A numeric literal is a form of literal, designed primarily for convenience in arithmetic calculation, whose value is restricted to Vn.

The syntax of numeric literal is that of exponential value, except that no leading sign is permitted. (See 11.1.3.)

numeric literal ::= mantissa [exponent]

The value of a numeric literal is derived from its spelling by direct application of the interpretation operation Ien (see 11.1.3.2).

The construct integer literal is used in other definitions within this manual.

integer literal ::= digit [digit] ...

The value of an integer literal is derived from its spelling by direct application of the interpretation operation Ii (see 11.1.3).

Examples of literals

Spelling	Value
"HELLO"	HELLO
"""HELLO"""	"HELLO"
1	1
0001	1
1.23	1.23
1E1	10
1E2	100
1.5E2	150
3.1415E3	3141.5
2.34E-2	.0234
.01E2	1
"0001"	0001

12.4 Storage References

A storage reference is a unique designator of a node of either Named Partition Storage or Named System Storage. Storage references may appear in contexts other than the expression atom context (for example, in the argument of the $DATA function or to the left of the = in the SET command). Whenever a storage reference appears in the expression atom context, its value is the content of the Value attribute of the designated storage node; in this context, if there is no storage node as designated by the storage reference, or if the storage node exists but its Value attribute is undefined, execution of the storage reference is considered to be in error, and it has no defined value.

Syntactically, there are three varieties of storage reference, as enumerated in the following syntax definitions.

```
name ::= | %     | | [digit] |
         | alpha |   [alpha]  | ...
```

[Port: Routines designed for portability should not have any lower-case letters in names. names with 9 or more characters should be distinguished on the basis of their first 8 characters. :Port]

```
storage reference ::= | local variable  |
                      | global variable |
                      | naked reference |

local variable ::= | name [(expression [,expression] ... )] |
                   | @ expression atom V local variable     |

global variable ::= | ^name [(expression [,expression] ... )] |
                    | @ expression atom V global variable     |

naked reference ::= | ^(expression [,expression] ... )     |
                    | @ expression atom V naked reference |
```

The expressions in parentheses are called "subscripts".

The syntactic form	Designates a node in
local variable	Named Partition Storage
global variable	Named System Storage
naked reference	Named System Storage

The LOCK command requires a name-only reference which excludes naked reference. variable name exists for this purpose.

```
variable name ::= | local variable  |
                  | global variable |
```

The algorithm by which an instance of storage reference designates a storage node consists of the following two steps.

1. A mapping, called here the Node Designation Mapping (described below), is executed, using the storage reference instance as its argument. The value of this mapping is a node designator (see 4.2).
2. If a storage node exists whose Name attribute contains the same string as the computed node designator, this storage node is unique and it is the designated storage node. If it does not exist, the resultant action depends on the context in which the storage reference appears.

Description of Node Designation Mapping

The Node Designation Mapping starts with an empty Node Designator Register and builds up the final node designator by a process of successive concatenation to the right of the content of the Node Designator Register.

1. Call the Node Designator Register "NDR". The first step depends on the syntactic form of the storage reference or variable name instance.

    a. If local variable, place the instance of name into the NDR.
    b. If global variable, place the instance of ^name into the NDR. (That is, the first character is "^".)
    c. If naked reference, place the content of the Naked Indicator into the NDR. If the Naked Indicator is undefined, the storage reference instance is erroneous.

2. If the storage reference or variable name instance is a local variable or global variable which contains no subscript, the Node Designation Mapping ends here. Otherwise, proceed to step 3.

3. Execute this step once for each subscript expression, proceeding in a left-to-right direction. This step contains the following two parts.

    a. First, evaluate the expression. Let V denote its value.
    b. Then concatenate ¢V to the right of the NDR.

4. This completes the Node Designation Mapping. If the argument is a storage reference, refer to 5.2 for a description of any effect on the Naked Indicator of executing a global variable or naked reference. If the argument is a variable name, there is no effect on the Naked Indicator.

[Port: MUMPS routines designed for portability may only execute subscripts whose values consist entirely of characters from the ASCII graphic subset. The length of each subscript is limited to 31 characters. When the subscript value satisfies the definition of numeric value (11.1.2),it is further subject to the portability restrictions for numeric value (11.1.2), and to the set of nonnegative numbers only.

There may be at most nine digits in the value of a numeric valued subscript expression. Accompanying each implementation should be a description of the interpreter's response to subscript expressions whose values are other than those just described. Interpreters which accept such subscript values should do so in a way which maintains consistency with the rest of this reference manual. :Port]

Examples of correct local variables

```
X          AB          A(3)       AB(4,5,X)    A(B(3))    A(1.5)
A(3,4)     AB(X,Y(Z),^A(3,4,5))    A(B(C(D(E,F(G))))) A(0.4)
A(1)       A(--1)      A(+1)      A("1")       A(1.0)     A(1278.56)
A(01)      A(1.5E1)    A(1.)      A(+"12A")    A("PART")  A("Name","Age")
A("SMITH","Jane")      A("124-79-6246")        A("City")
```

Examples of nonportable storage references

```
A(-2)      A(-234)     A("-.3")
A("A|VERY|LONG|NONPORTABLE|SUBSCRIPT")
A(1.5E27)
```

Examples of Naked References

Prior content of Naked Indicator	Storage Reference	Node designator	Subsequent content of Naked Indicator
undefined	^A(1)	^Ac1	^A
^A	^(2)	^Ac2	^A
^A	^(3,4)	^Ac3c4	^Ac3
^Ac3	^(5)	^Ac3c5	^Ac3
^Ac3	^B(3,5)	^Bc3c5	^Bc3
^Bc3	^X	^X	undefined
undefined	^(2)	error	undefined
undefined	^Y	^Y	undefined
undefined	^X(5,6,7,8)	^Xc5c6c7c8	^Xc5c6c7
^Xc5c6c7	^(9)	^Xc5c6c7c9	^Xc5c6c7
^Xc5c6c7	^(9,-2)	^Xc5c6c7c9c-2 (*)	^Xc5c6c7c9 (*)
^Xc5c6c7c9	^("+")	^Xc5c6c7c9c+ (*)	^Xc5c6c7c9 (*)

(*)Note: Some implementations may properly consider this to be erroneous; if they do not, this is the result they should show.

12.5 Unary Operators

The three operators '(not), +(plus), and -(minus) are called unary operators. Their effects are defined as follows.

The value of 'truth-value expression atom is the complement of the value of truth-value expression atom. That is, where T is a truth-value expression atom,

$$'T = \begin{cases} 1 & \text{if the value of T is 0} \\ 0 & \text{if the value of T is 1} \end{cases}$$

The value of +numeric expression atom is the same as the value of numeric expression atom. That is, + is simply a way of forcing execution of the numeric interpretation In.

The value of -numeric expression atom is the negative of the value of numeric expression atom. Let V be the value of numeric expression atom. The value of -numeric expression atom is the result of applying In to the string -V. (Simplification of double negatives occurs in the sign reduction rules of 11.1.3.1.)

Let unary operator denote any of the three unary operators. The portion of the syntax description of expression atom referring to unary operators reads as follows.

expression atom ::= unary operator expression atom

This recursive formulation is chosen over an iterative formulation expressed informally by

[unary operator] ... expression atom

in order to make explicit that the order of evaluation of unary operators is right-to-left.

Examples of unary operators

Expression	Value	
+"A"	0	
+-"A"	0	
-"5.0"	-5	
'"5.0"	0	
'(1+2E-10-1)	1	(possibly)
'(1-1+2E-10)	0	(definitely)
3--5	8	
3-0--5	8	
3---5	-2	
3--'"A"	4	

APPENDIX A

Table of ASCII Characters

The character notation is that used in ANS X3.4-1968. The code values are those which appear as values of the $ASCII function and as arguments of the $CHAR function.

Code	Character	Code	Character	Code	Character	Code	Character
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

APPENDIX B

Index of Syntactic Types

Page numbers refer to principal definitions.

APPENDIX C

Index of Technical Terms

MUMPS LANGUAGE STANDARD

Part I: MUMPS Language Specification

MUMPS Development Committee

The reader is hereby notified that the language specifications contained in this Standard have been approved by the MUMPS Development Committee, but that they may be partial specifications which rely on information appearing in many parts of the MUMPS specifications. The specifications are dynamic in nature, and the changes reflected by these approved releases may not correspond with the latest specifications available.

Because of the evolutionary nature of MUMPS specifications the reader is further reminded that changes are likely to occur in the specifications released herein prior to a complete republication of MUMPS specifications.

American National Standard

An American National Standard implies a consensus of those substantially concerned with its scope and provisions. An American National Standard is intended as a guide to aid the manufacturer, the consumer, and the general public. The existence of an American National Standard does not in any respect preclude anyone, whether he has approved the standard or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standard. American National Standards are subject to periodic review and users are cautioned to obtain the latest editions.

Caution Notice: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to reaffirm, revise, or withdraw this standard no later than five years from the date of publication. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

Developed and Published by

MUMPS Development Committee

This edition of the ANS MUMPS Language Standard is published by the MUMPS Users' Group.

Printed in the United States of America

Table of Contents

1. Overview of MUMPS Language Specification

1.1 Organization of This Document

This document describes the MUMPS language at two levels of detail. Subsection 1.2 gives an overview of the prominent features of the language, intended for the reader who is already familiar with at least one existing dialect. Section 3 describes the static syntax of the language. The distinction between "static" and "dynamic" syntax is as follows. The static syntax describes the sequence of characters in a program as it appears on a tape in program interchange or on a listing. The dynamic syntax describes the sequence of characters actually encountered by an interpreter during execution of the program. The dynamic syntax takes into account transfers of control and values produced by indirection. Section 2 describes the metalanguage used for the static syntax.

1.2 Summary of the Language

1.2.1 Character Set

The character set which is used for the interchange of MUMPS programs and data is the seven-bit USA Standard Code for Information Interchange (ASCII) defined by ANSI X3.4-1968. Programs may be written entirely with the common 64-character subset of ASCII. The character collating sequence is the same as the numeric sequence of the ASCII character codes.

1.2.2 Routine Structure

A MUMPS routine consists of a sequence of lines. For purposes of transfer of control, lines may be optionally labeled. A label is either a conventional MUMPS name (an initial letter or % followed by alphanumerics), or it is an integer literal.

1.2.3 Program Punctuation

The following special characters may occur in programs.

Unary Arithmetic Operators

+	plus
-	negate

Unary Logical Operator

'	not

Binary Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division
\	division with integer quotient
#	modulo

Binary Relational Operators

<	numeric less than
'<	numeric greater than or equal
>	numeric greater than
'>	numeric less than or equal
=	string identity
'=	string nonidentity
[	string contains
'[	string not-contains
]	string follows
']	string not-follows
?	string pattern match
'?	string pattern nonmatch

Binary Logical Operators

&	and
'&	nand
!	or
'!	nor

Binary String Operator

_	concatenation

Delimiters

,	argument separation, subscript separation
=	value assignment
:	post-conditional expression, subargument separation
( )	grouping
@	indirection
"	string literals
.	decimal point in numeric literals
^	preceding routine name in DO, GOTO
E	preceding exponent in numeric literals
;	comment
space	separating command words

Prefixes

^	global variable names
$	functions, special variable names
%	available in names of the programmer's choice

1.2.4 Data Types

Arithmetic operations are performed on strings and produce numeric values, which are special cases of strings. This approach to the standard specification does not preclude the use of multiple data representations within an implementation of the standard.

Any string value may enter into an arithmetic operation; there is a uniform rule for interpreting a string as a number. Certain operations deal with integer values, which are special cases of numeric values; the latter may contain decimal fractions. There is a uniform rule for interpreting any number (and, by inference, any string) as an integer.

Certain other operations deal with truth values, which are special cases of numeric values. There are two truth values: 0 and 1. The integer value 0 is the truth value 0. The integer value 1 is the truth value 1. All other numeric values are interpreted as the truth value 1. The truth value 0 denotes False; the truth value 1 denotes True.

1.2.5 Precedence of Operators

All binary operators are at the same level of precedence. Application of unary operators precedes application of binary operators.

1.2.6 Commands

At present, the standard contains only program-mode ("indirect") commands, of which the following are defined.

BREAK	provides an access point within the standard for non-standard programming and debugging aids.
CLOSE	releases one or more devices from ownership.
DO	provides a generalized subroutine call.
ELSE	permits conditional execution.
FOR	controls repetitive execution over a set of values of a variable.
GOTO	provides a generalized transfer of control.
HALT	terminates execution.
HANG	suspends execution for a specified period of time.
IF	permits conditional execution.
KILL	controls the elimination of specified variables and their values.
LOCK	provides a generalized interlock facility for coordinating concurrent processes.
OPEN	obtains ownership of one or more devices.
QUIT	defines an exit point of FOR or DO.
READ	specifies data input.
SET	assigns values to variables.
USE	designates a specific device for input and output.
VIEW	provides an access point within the standard for the examination of machine-dependent information.
WRITE	specifies data output.
XECUTE	permits execution of strings arising from the expression evaluation process.
Z	reserved for implementation-specific extensions.

All other command words, except those beginning with Z, are reserved.

1.2.7 Functions

The following functions are currently specified.

$ASCII	selects a character of a string and returns its code as an integer.
$CHAR	translates a set of integers into a string of character whose codes are those integers.
$DATA	returns an integer specifying whether a defined value and/or pointer of a named variable exists.
$EXTRACT	returns a character or substring of a string expression, selected by position number.
$FIND	returns an integer specifying the end position of a specified substring within a string.
$JUSTIFY	returns the value of an expression, right-justified within a field of specified size.
$LENGTH	returns the length of a string.
$NEXT	returns the lowest numeric subscript value on the same level, but numerically higher than the last subscript of the named global or local variable.
$PIECE	returns a string between two specified occurrences of a specified substring within a specified string.
$RANDOM	returns a pseudo-random number in a specified interval.
$SELECT	returns the value of one of several expressions in a list, selected by the truth values in a second list of expressions.
$TEXT	returns the text content of a specified line of the routine in which the function appears.
$VIEW	reserved for implementation-specific methods of obtaining machine-dependent data.
$Z	reserved for definition of implementation-specific functions.

All other initial letters of function names are reserved.

1.2.8 Special Variables

The following special variables are specified.

$HOROLOG provides the date and time in a single, two-part value.

$IO identifies the currently assigned I/O device.

$JOB has an integer value which uniquely identifies the process which evaluates it.

$STORAGE provides the number of unused characters which remain in a routine's partition.

$TEST makes available the truth value determined by the IF command and by the OPEN, LOCK, and READ with timeouts.

$X gives the horizontal cursor position on the current device.

$Y gives the line number on the current device.

$Z reserved for implementation-specific definitions.

All other initial letters of special variable names are reserved.

2. Static Syntax Metalanguage

The primitives of the metalanguage are the ASCII characters and the metalanguage operators ::= (definition), [] (option), || (grouping), ... (optional indefinite repetition), L (list), and V (value).

In general, defined syntactic objects will have designations which are underlined names spelled with lower-case letters, e.g., name, expr, etc. Concatenation of syntactic objects is expressed in the static syntax by horizontal juxtaposition. Choice is expressed by vertical juxtaposition. The ::= symbol denotes a syntactic definition. An optional element is enclosed in square brackets [], and three dots ... denote that the previous element is optionally repeated any number of times. The definition of name, for example, is written:

$$\underline{\text{name}} \quad ::= \quad \left| \begin{matrix} \% \\ \underline{\text{alpha}} \end{matrix} \right| \quad \left[\begin{matrix} \underline{\text{digit}} \\ \underline{\text{alpha}} \end{matrix} \right] \quad \ldots$$

The vertical bars are used only to group elements for repetition or to make a group of elements more readable. When there is any danger of confusing the square brackets in the metalanguage with the ASCII graphics [and], special care is taken to avoid this. Normally, the square brackets will stand for the metalanguage symbols.

The unary metalanguage operator L denotes a list of one or more occurrences of the syntactic object immediately to its right, with commas between each pair of occurrences. Thus,

L name is equivalent to name [, name] ...

The binary metalanguage operator V, used in the specification of indirection, places the constraint on the expratom to its left that it must have a value which satisfies the syntax of the syntactic object to its right. For example, one might define the syntax of a hypothetical EXAMPLE command with its argument list by

examplecommand ::= EXAMPLE ␣ L exampleargument

where

$$\underline{\text{exampleargument}} \quad ::= \left| \begin{matrix} \underline{\text{expr}} \\ \\ @\ \underline{\text{expratom}}\ \underline{\text{V}}\ \underline{\text{L}}\ \underline{\text{exampleargument}} \end{matrix} \right|$$

This says that, after evaluation of indirection, the command argument list consists of any number of exprs separated by commas. In the static syntax (i.e., prior to evaluation of indirection), occurrences of @ expratom may stand in place of nonoverlapping sublists of command arguments. Usually, the text accompanying a syntax description incorporating indirection will describe the syntax after all occurrences of indirection have been evaluated.

3. Static Syntax

3.1 Basic Alphabet

The routine, which is the object whose static syntax is being described in Section 3, is a string made up of the following 98 symbols.

The 95 ASCII graphics, including SP (space)
The line-start symbol ls
The end-of-line symbol eol
The end-of-routine symbol eor

In program interchange, the following ASCII characters are used in place of ls, eol, and eor.

ls:	SP
eol:	CR LF
eor:	CR FF

When a program is stored internally, the standard does not specify what forms ls, eol, and eor take. They may, in fact, be expressed by means other than characters in the program. When a program is entered from a keyboard, the standard does not specify what operator procedures correspond to ls, eol, or eor.

The syntactic types graphic, alpha, and digit are defined here informally in order to save space.

graphic ::= any of the class of 95 ASCII graphics, including SP (space), represented by ␣ or SP.

alpha ::= any of the class of 52 upper and lower case letters: A-Z, a-z.

digit ::= any of the class of 10 digits: 0-9.

3.2 Expression Atom expratom

The expression, expr, is the syntactic element which denotes the execution of a value-producing calculation; it is defined in 3.3. The expression atom, expratom, is the basic value-denoting object of which expressions are built; it is defined here.

```
expratom ::=  | lvn                  |
              | gvn                  |
              | svn                  |
              | function             |
              | numlit               |
              | strlit               |
              | ( expr )             |
              | unaryop expratom     |
```

```
unaryop ::=  | ' |     (Note: apostrophe)
             | + |
             | - |     (Note: hyphen)
```

3.2.1 Name name

```
name ::=  | % alpha | [ digit ]  ...
          |         | [ alpha ]
```

3.2.2 Local Variable Name lvn

```
lvn ::=  | name [( L expr )]      |
         | @ expratom V lvn       |
```

A local variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. An unsubscripted occurrence of lvn may carry a different value from any subscripted occurrence of lvn.

3.2.3 Global Variable Name gvn

```
gvn  ::=  | ^( L expr )                  |
          | ^name [( L expr )]           |
          | @ expratom V gvn              |
```

The prefix ^ uniquely denotes a global variable name. A global variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. There is permitted an abbreviated form of subscripted gvn, called the "naked reference", in which the name and an initial (possibly empty) sequence of subscripts is absent but implied by the value of the "naked indicator". An unsubscripted occurrence of gvn may carry a different value from any subscripted occurrence of gvn.

Every executed occurrence of gvn affects the naked indicator as follows. If, for any positive integer m, the gvn has the nonnaked form N(v1, v2, ..., vm), then the m-tuple N, v1, v2, ..., vm-1, is placed into the naked indicator when the gvn reference is made. A subsequent naked reference of the form

^(s1, s2, ..., si) (i positive)

results in a global reference of the form

N(v1, v2, ..., vm-1, s1, s2, ..., si)

after which the m+i-1-tuple N, v1, v2, ..., si-1 is placed into the naked indicator. Prior to the first executed occurrence of a nonnaked form of gvn, the value of the naked indicator is undefined. It is erroneous for the first executed occurrence of gvn to be a naked reference.

Two types of global references leave the naked indicator undefined.

a. A nonnaked reference without subscripts.
b. A nonnaked reference of the form, or a naked reference resulting in the form

   N(v1, v2, ..., vn) (n positive)

   for which $D(N(v1, v2, ..., vn-1)) < 10.
   (If n=1, read: $D(N) < 10.)

The effect on the naked indicator described above occurs regardless of the context in which gvn is found; in particular, an assignment of a value to a global variable with the command SET gvn = expr does not affect the value of the naked indicator until after the right-side expr has been evaluated. The effect on the naked indicator of any gvn within the right-side expr will precede the effect on the naked indicator of the left-side gvn.

For convenience, glvn is defined so as to be satisfied by the syntax of either gvn or lvn.

```
glvn ::= | gvn |
         | lvn |
```

3.2.4 Numeric Literal numlit

The integer literal syntax, intlit, which is a nonempty string of digits, is defined here.

```
intlit ::= digit [ digit ] ...
```

The numeric literal numlit is defined as follows.

```
numlit ::= mant [ exp ]

 mant  ::= | intlit [ . intlit ] |
           | . intlit              |

  exp  ::= E [ + ] intlit
             [ - ]
```

The value of the string denoted by an occurrence of numlit is defined in the following two subsections.

3.2.4.1 Numeric Data Values

All variables, local, global, and special, have values which are either defined or undefined. If defined, the values may always be thought of and operated upon as strings. The set of numeric values is a subset of the set of all data values.

Only numbers which may be represented with a finite number of decimal digits are representable as numeric values. A data value has the form of a number if it satisfies the following restrictions.

a. It may contain only digits and the characters "-" and ".".
b. At least one digit must be present.
c. "." occurs at most once.
d. The number zero is represented by the one-character string "0".
e. The representation of each positive number contains no "-".
f. The representation of each negative number contains the character "-" followed by the representation of the positive number which is the absolute value of the negative number. (Thus, the following restrictions describe positive numbers only.)
g. The representation of each positive integer contains only digits and no leading zero.
h. The representation of each positive number less than 1 consists of a "." followed by a nonempty digit string with no trailing zero. (This is called a "fraction".)
i. The representation of each positive noninteger greater than 1 consists of the representation of a positive integer (called the "integer part" of the number) followed by a fraction (called the "fraction part" of the number).

Note that the mapping between representable numbers and representations is one-to-one. An important result of this is that string equality of numeric values is a necessary and sufficient condition of numeric equality.

3.2.4.2 Meaning of numlit

Note that numlit denotes only nonnegative values. The process of converting the spelling of an occurrence of numlit into its numeric data value consists of the following steps.

a. If the mant has no ".", place one at its right end.
b. If the exp is absent, skip step c.
c. If the exp has a plus or has no sign, move the "." a number of decimal digit positions to the right in the mant equal to the value of the intlit of exp, appending zeros to the right of the mant as necessary. If the exp has a minus sign, move the "." a number of decimal digit positions to the left in the mant equal to the value of the intlit of exp, appending zeros to the left of the mant as necessary.
d. Delete the exp and any leading or trailing zeros of the mant.
e. If the rightmost character is ".", remove it.
f. If the result is empty, make it "0".

3.2.5 Numeric Interpretation of Data

Certain operations, such as arithmetic, deal with the numeric interpretations of their operands. The numeric interpretation is a mapping from the set of all data values onto the set of all numeric values, described by the following algorithm. Note that the numeric interpretation maps numeric values onto themselves.

> (Note: The "head" of a string is defined to be a substring which contains all of the characters of the string to the left of a given point and none of the characters of the string to the right of that point. A head may be empty or it may be the entire string.)

Consider the argument to be the string S.

First, apply the following sign reduction rules to S as many times as possible, in any order.

a. If S is of the form + T, then remove the +. (Shorthand: + T → T)
b. - + T → - T
c. - - T → T

Second, apply one of the following, as appropriate.

a. If the leftmost character of S is not "-", form the longest head of S which satisfies the syntax description of numlit. Then apply the algorithm of 3.2.4.2 to the result.
b. If S is of the form - T, apply step a. above to T and append a "-" to the left of the result. If the result is "-0", change it to "0".

The "numeric expression" numexpr is defined to have the same syntax as expr. Its presence in a syntax description serves to indicate that the numeric interpretation of its value is to be taken when it is executed.

numexpr ::= expr

3.2.5.1 Integer Interpretation

Certain functions deal with the integer interpretations of their arguments. The integer interpretation is a mapping from the set of all data values onto the set of all integer values, described by the following algorithm.

First, take the numeric interpretation of the argument. Then remove the fraction, if present. If the result is empty or "-", change it to "0".

The "integer expression" intexpr is defined to have the same syntax as expr. Its presence in a syntax definition serves to indicate that the integer interpretation of its value is to be taken when it is executed.

intexpr ::= expr

3.2.5.2 Truth-Value Interpretation

The truth-value interpretation is a mapping from the set of all data values onto the two integer values 0 and 1, described by the following algorithm. Take the numeric interpretation. If the result is not "0", make it "1".

The "truth-value expression" tvexpr is defined to have the same syntax as expr. Its presence in a syntax definition serves to indicate that the truth-value interpretation of its value is to be taken when it is executed.

tvexpr ::= expr

3.2.6 String Literal strlit

Let nonquote temporarily be defined as any of the class of 94 graphics, excluding the quote symbol.

strlit ::= " [" " / nonquote] ... "

In words, a string literal is bounded by quotes and contains any string of graphics, except that when quotes occur inside, they occur in adjacent pairs. Each such adjacent quote pair denotes a single quote in the value denoted by strlit, whereas any other graphic between the bounding quotes denotes itself. An empty string is denoted by exactly two quotes.

3.2.7 Special Variable Name svn

Special variables are denoted by the prefix $ followed by one of a designated list of names. The following special variables are defined.

Syntax	Definition
$H[OROLOG]	$H gives date and time with one access. Its value is D,S where D is an integer value counting days since an origin specified below, and S is an integer value modulo 86,400 counting seconds. The value of $H for the first second of December 31, 1840 after midnight is defined to be 0,0. S increases by 1 each second and S clears to 0 with a carry into D on the tick of midnight.
$I[O]	$I identifies the current I/O device. See 3.6.2 and 3.6.16.
$J[OB]	Each executing MUMPS process has its own job number, a positive integer which is the value of $J. The job number of each process is unique to that process within a domain of concurrent processes defined by the implementor. $J is constant throughout the active life of a process.
$S[TORAGE]	Each implementation must return for the value of $S an integer which is the number of characters of free space available for use. The method of arriving at the value of $S is not part of the standard.
$T[EST]	$T contains the truth value computed from the execution of the most recent IF command containing an argument, or an OPEN, LOCK, or READ with a timeout.
$X	$X has a nonnegative integer value which approximates the value of a carriage or horizontal cursor position on the current line as if the current I/O device were an ASCII terminal. It is initialized to zero by input or output of control functions corresponding to CR or FF; input or output of each graphic adds 1 to $X. See 3.5.5 and 3.6.16.
$Y	$Y has a nonnegative integer value which approximates the line number on the current I/O device as if it were an ASCII terminal. It is initialized to zero by input or output of control functions corresponding to FF; input or output of control functions corresponding to LF adds 1 to $Y. See 3.5.5 and 3.6.16.
$Z[unspecified]	Z is the initial letter reserved for defining nonstandard special variables. The requirement that $Z be used permits the unused initial letters to be reserved for future enhancement of the standard without altering the execution of existing programs which observe the rules of the standard.

3.2.8 Functions function

Functions are denoted by the prefix $ followed by one of a designated list of names, followed by a parenthesized argument list. Any of the following specifications satisfies the definition of function.

$A[SCII](expr)

produces an integer value as follows:

a. -1 if the value of expr is the empty string.
b. Otherwise, the decimal equivalent of the ASCII code of the leftmost character of the value of expr.

$A[SCII](exprl , intexpr2)

is similar to $A(exprl) except that it works with the intexpr2th character of exprl instead of the first. Formally, $A(exprl,intexpr2) is defined to be $A($E(exprl,intexpr2)).

$C[HAR](L intexpr)

returns a string whose length is the number of argument expressions which have integer values in the closed interval [0,127]. Each intexpr in that interval maps into the ASCII character whose code is the value of intexpr; this mapping is order-preserving. Each negative-valued intexpr maps into no character in the value of $C. Any intexpr whose value is greater than 127 is erroneous.

$D[ATA](glvn)

returns a nonnegative integer which is a characterization of the variable named. The value of the integer is p+d, where:

d = 1 if the named variable has a defined value;
d = 0 otherwise;
p = 10 if either:

a. The named variable exists and contains no subscripts, and there exists (or did exist and was killed) a subscripted variable with the same name, or
b. The named variable exists and contains n subscripts, and there exists (or did exist and was killed) a subscripted variable with m > n subscripts whose first n subscript values are the same as the values of those in the named variable;

p = 0 otherwise.

$E[XTRACT](exprl , intexpr2)

returns the intexpr2th character of the value of exprl. That is, let m be the value of intexpr2. If m is less than 1 or greater than $L(exprl), the value of $E is the empty string. Otherwise, the value of $E is the mth character of the value of exprl. (1 corresponds to the leftmost character; $L(exprl) corresponds to the rightmost character.)

$E[XTRACT](exprl , intexpr2 , intexpr3)

returns the string between positions intexpr2 and intexpr3 of the value of exprl. Let m be the value of intexpr2 and let n be the value of intexpr3. The following cases are defined:

a. m > n. Then the value of $E is the empty string.
b. m = n. $E(exprl,m,n) = $E(exprl,m).
c. m < n ≤ $L(exprl). $E(exprl,m,n) = $E(exprl,m) concatenated with $E(exprl,m+1,n).
d. m < n and $L(exprl) < n. $E(exprl,m,n) = $E(exprl,m,$L(exprl)).

$F[IND](exprl , expr2)

searches for the leftmost occurrence of the value of expr2 in the value of exprl. If none is found, $F returns zero. If one is found, the value returned is the integer representing the number of the character position immediately to the right of the rightmost character of the found occurrence of expr2 in exprl.

$F[IND](exprl , expr2 , intexpr3)

Let m be the value of intexpr3. If m is less than 1, the value of $F is $F(exprl,expr2). Otherwise, $F begins the search at the mth position of exprl. If no instance of expr2 is found, $F returns zero; otherwise, $F(A,B,m) = $F($E(A,m,$L(A)),B) + m - 1.

$J[USTIFY] (exprl , intexpr2)

returns the value of exprl right-justified in a field of intexpr2 spaces. Let m be $L(exprl) and n be the value of intexpr2. The following cases are defined:

a. m ≥ n. Then the value returned is exprl.
b. Otherwise, the value returned is S(n-m) concatenated with exprl, where S(x) is a string of x spaces.

$J[USTIFY] (numexprl , intexpr2 , intexpr3)

returns an edited form of the number numexprl. Let R be the value of numexprl after rounding to intexpr3 fraction digits, including possible trailing zeros. (If intexpr3 is zero, R contains no decimal point.) The value returned is $J(R,intexpr2). Negative values of intexpr3 are reserved for future extensions of the $JUSTIFY function.

$L[ENGTH](expr)

returns an integer which is the number of characters in the value of expr. $L of the empty string is zero.

$N[EXT](glvn)

returns an integer which is a subscript value. Only subscripted forms of lvn and gvn are permitted. Let lvn or gvn be of the form Name(s1, s2, ..., sn) where sn has an integer value $\geq$ -1. If it exists, the integer value t is returned, where:

a. t is greater than sn, and
b. t is the lowest number such that $D(Name(s1, s2, ..., sn-1, t) is not zero.

If such a t does not exist, -1 is returned. It is an error if sn has a noninteger value or an integer value less than -1.

$P[IECE](expr1 , expr2 , intexpr3 [, intexpr4])

is defined here with the aid of a function, NF, which is used only for definitional purposes called "find the position number following the mth occurrence".

NF(S, D, m) is defined, for strings S, D, and integer m, as follows:

When m $\leq$ 0, the result is zero.

When D is not a substring of S, i.e., when $F(S, D) = 0, then the result is $L(S) + $L(D) + 1.

Otherwise, NF(S, D, 1) = $F(S, D).

For m > 1,

NF(S, D, m) = NF($E(S, $F(S, D), $L(S)), D, m-1) + $F(S, D) - 1.

That is, NF generalizes $F to give the position number of the character to the right of the mth occurrence of the string D in S.

$P[IECE](expr1 , expr2 , intexpr3)

Let expr1, expr2 be the strings S, D. Let intexpr3 be the integer m. $P(S, D, m) returns the substring of S bounded by but not including the m - 1th and the mth occurrences of D.

$P(S, D, m) = $E(S, NF(S, D, m-1), NF(S, D, m) - $L(D) - 1).

$P[IECE](expr1 , expr2 , intexpr3 , intexpr4)

Let intexpr4 be the integer n. $P(S, D, m, n) returns the substring of S bounded on the left but not including the m - 1th occurrence of D in S, and bounded on the right but not including the nth occurrence of D in S.

$P(S, D, m, n) = $E(S, NF(S, D, m-1), NF(S, D, n) - $L(D) - 1).

Note that $P(S, D, m, m) = $P(S, D, m).

$R[ANDOM](intexpr)

returns a random or pseudo-random integer uniformly distributed in the closed interval [0, intexpr - 1]. If the value of intexpr is less than 1, an error will occur.

$S[ELECT](L |tvexpr:expr|)

returns the value of the leftmost expr whose corresponding tvexpr is true. The process of evaluation consists of evaluating the tvexprs, one at a time in left-to-right order, until the first one is found whose value is true. The expr corresponding to this tvexpr (and no other) is evaluated and this value is made the value of $S. An error will occur if all tvexprs are false. Since only one expr is evaluated at any invocation of $S, that is the only expr which must have a defined value.

$T[EXT](| + intexpr |)
         | lineref |

returns a string whose value is the content of the line of this routine specified by the argument. Specifically, the entire line, with ls replaced by one SP and eol deleted, is returned.

If the argument of $T is a lineref, the line denoted by the lineref is specified. If the argument is + intexpr, the intexprth line of the routine is specified. The first line is numbered 1; an error will occur if the value of intexpr is less than 1.

If no such line as that specified by the argument exists, an empty string is returned. If the line specification is ambiguous, the results are not defined.

$V[IEW](unspecified)

makes available to the implementor a call for examining machine-dependent information. It is to be understood that programs containing occurrences of $V may not be portable.

$Z[unspecified](unspecified)

is the name reserved for defining escapes to nonstandard functions. This requirement permits the unused function names to be reserved for future use.

3.2.9 Unary Operator unaryop

There are three unary operators: ' (not), + (plus), and - (minus).

Not inverts the truth value of the expratom immediately to its right. The value of 'expratom is 1 if the truth-value interpretation of expratom is 0; otherwise its value is 0. Note that '' performs the truth-value interpretation.

Plus is merely an explicit means of taking a numeric interpretation. The value of + expratom is the numeric interpretation of the value of expratom.

Minus negates the numeric interpretation of expratom. The value of - expratom is the numeric interpretation of -N, where N is the value of expratom.

Note that the order of application of unary operators is right-to-left.

3.3 Expressions expr

Expressions are made up of expression atoms separated by binary string, arithmetic, or truth-valued operators.

```
expr      ::=  expratom [ exprtail ] ...

exprtail  ::=  | [binaryop | ['] truthop]  expratom |
               |          ['] ? pattern              |

binaryop  ::=  | _ |        (Note: underscore)
               | + |
               | - |        (Note: hyphen)
               | * |
               | / |
               | \ |
               | # |

truthop   ::=  | relation |
               | logicalop |

relation  ::=  | = |
               | < |
               | > |
               | [ |
               | ] |

logicalop ::=  | & |
               | ! |
```

The order of evaluation is as follows:

a. Evaluate the left-hand expratom.
b. If an exprtail is present immediately to the right, evaluate its expratom or pattern and apply its operator.
c. Repeat step b. as necessary, moving to the right.

In the language of operator precedence, this sequence implies that all binary string, arithmetic, and truth-valued operators are at the same precedence level and are applied in left-to-right order.

Any attempt to evaluate an expratom containing an lvn, gvn, or svn with an undefined value is erroneous.

3.3.1 Arithmetic Binary Operators

The binary operators + - * / \ # are called the arithmetic binary operators. They operate on the numeric interpretations of their operands, and they produce numeric (in one case, integer) results.

+ produces the algebraic sum.

- produces the algebraic difference.

* produces the algebraic product.

/ produces the algebraic quotient. Note that the sign of the quotient is negative if and only if one argument is positive and one argument is negative. Division by zero is erroneous.

\ produces the integer interpretation of the result of the above division.

produces the value of the left argument modulo the right argument. It is defined only for nonzero values of its right argument, as follows.
A # B = A - (B * floor(A/B))
where floor(x) = the largest integer $\leq$ x.

3.3.2 Relational Operators

The operators = < >] [produce the truth value 1 if the relation between their arguments which they express is true, and 0 otherwise. The dual operators 'relation are defined by:

A 'relation B has the same value as '(A relation B)

3.3.2.1 Numeric Relations

The inequalities > and < operate on the numeric interpretations of their operands; they denote the conventional algebraic "greater than" and "less than".

3.3.2.2 String Relations

The relations =] [do not imply any numeric interpretation of either of their operands.

The relation = tests string identity. If the operands are not known to be numeric and numeric equality is to be tested, the programmer may apply an appropriate unary operator to the nonnumeric operands. If both arguments are known to be in numeric form (as would be the case, for example, if they resulted from the application of any operator except _), application of a unary operator is not necessary. The uniqueness of the numeric representation guarantees the equivalence of string and numeric equality when both operands are numeric. Note, however, that the division operator / may produce inexact results, with the usual problems attendant to inexact arithmetic.

The relation [is called "contains". A [B is true if and only if B is a substring of A; that is, A [B has the same value as ''$F(A,B). Note that the empty string is a substring of every string.

The relation] is called "follows". A] B is true if and only if A follows B in the conventional ASCII collating sequence, defined here. A follows B if and only if any of the following is true.

a. B is empty and A is not.
b. Neither A nor B is empty, and the leftmost character of A follows (i.e., has a numerically greater ASCII code than) the leftmost character of B.
c. There exists a positive integer n such that A and B have identical heads of length n, (i.e., $E(A,1,n) = $E(B,1,n)) and the remainder of A follows the remainder of B (i.e., $E(A,n+1,$L(A)) follows $E(B,n+1,$L(B))).

3.3.3 Pattern Match

The pattern match operator ? tests the form of the string which is its left-hand operand. S ? P is true if and only if S is a member of the class of strings specified by the pattern P.

A pattern is a concatenated list of pattern atoms.

$$\underline{\text{pattern}} \quad ::= \quad \left| \begin{array}{l} \underline{\text{patatom}} \ [\ \underline{\text{patatom}} \ ] \ \ldots \\ \\ @ \ \underline{\text{expratom}} \ \underline{V} \ \underline{\text{pattern}} \end{array} \right|$$

Assume that pattern has n patatoms. S ? pattern is true if and only if there exists a partition of S into n substrings

$$S = S\underline{1} \ S\underline{2} \ \ldots \ S\underline{n}$$

such that there is a one-to-one order-preserving correspondence between the Si and the pattern atoms, and each Si "satisfies" its respective pattern atom. Note that some of the Si may be empty.

Each pattern atom consists of a pattern code patcode or a string literal strlit, preceded either by an integer literal intlit multiplier or by the indefinite multiplier ".".

```
patatom  ::=  | intlit |  | strlit  |
              |   .    |  | patcode |

              | C |
              | N |
              | P |
patcode  ::=  | A |  ...
              | L |
              | U |
              | E |
```

Each patcode is satisfied by any single character in the union of the classes of characters represented, each class denoted by its own patcode letter, as follows.

C	33 Control characters, including DEL
N	10 Numeric characters
P	33 Punctuation characters, including SP
A	52 Alphabetic characters
L	26 Lower-case alphabetic characters
U	26 Upper-case alphabetic characters
E	Everything (the Entire set of characters)

The strlit is satisfied by, and only by, the value of strlit.

If the indefinite multiplier "." is present, patatom is satisfied by a concatenation of any number of strings (including none), each of which satisfies the patcode or strlit following the multiplier.

If the intlit multiplier is present, patatom is satisfied by a concatenation of exactly intlit strings, each of which satisfies the patcode or strlit following the multiplier. In particular, if the value of intlit is zero, the corresponding Si is empty.

The dual operator '? is defined by:

```
A '? B = '(A ? B)
```

3.3.4 Logical Operators

The operators ! and & are called logical operators. (They are given the names "or" and "and", respectively.) They operate on the truth-value interpretations of their arguments, and they produce truth-value results.

$$A\ !\ B = \begin{cases} 0 \text{ if both A and B have the value 0} \\ 1 \text{ otherwise} \end{cases}$$

$$A\ \&\ B = \begin{cases} 1 \text{ if both A and B have the value 1} \\ 0 \text{ otherwise} \end{cases}$$

The dual operators '& and '! are defined by:

```
A '& B = '(A & B)
A '! B = '(A ! B)
```

3.3.5 Concatenation Operator

The underscore symbol _ is the concatenation operator. It does not imply any numeric interpretation. The value of A _ B is the string obtained by concatenating the values of A and B, with A on the left.

3.4 Routines

The routine is the unit of program interchange. In program interchange, each routine begins with its routinehead, which contains the identifying routinename, and the routinehead is followed by the routinebody, which contains the executed code. The routinehead is not part of the executed code.

routine ::= routinehead routinebody

routinehead ::= routinename eol

routinename ::= name

The routinebody is a sequence of lines terminated by the eor. Each line ends with eol, starts with ls optionally preceded by a label, and may contain zero or more commands (separated by single spaces) between ls and eol. Any line may end with a comment immediately preceding the eol.

routinebody ::= line [line] ... eor

line ::= [label] ls [command [␣ command] ... [␣ comment] | comment] eol

label ::= | name | intlit |

ls ::= SP (one space)

eol ::= CR LF (two control characters)

eor ::= CR FF (two control characters)

Each occurrence of a label to the left of ls in a line is called a "defining occurrence" of label. No two defining occurrences of label may have the same spelling in one routinebody.

3.5 General command Rules

Every command starts with a "command word" which dictates the syntax and interpretation of that command instance. The standard contains the following command words.

B[REAK]
C[LOSE]
D[O]
E[LSE]
F[OR]
G[OTO]
H[ALT]
H[ANG]
I[F]
K[ILL]
L[OCK]
O[PEN]
Q[UIT]
R[EAD]
S[ET]
U[SE]
V[IEW]
W[RITE]
X[ECUTE]
Z[unspecified]

Unused initial letters of command words are reserved for future enhancement of the standard.

The formal definition of the syntax of command is a choice from among all of the individual command syntax definitions of 3.6.

command	::=	syntax of BREAK command syntax of CLOSE command . . . syntax of XECUTE command

Any implementation of the language must be able to recognize both the initial letter abbreviation and the full spelling of each command word. When two command words have a common initial letter, their argument syntaxes uniquely distinguish them.

For all commands allowing multiple arguments, the form

command word arg1, arg2 ...

is equivalent in execution to

command word arg1 command word arg2 ... .

3.5.1 Post Conditionals

All commands except ELSE, FOR, and IF may be made conditional as a whole by following the command word immediately by the post-conditional postcond.

postcond ::= [: tvexpr]

If the tvexpr is either absent or present and true, the command is executed. If the tvexpr is present and false, the command word and its arguments are passed over without execution.

The postcond may also be used to conditionalize the arguments of DO, GOTO, and XECUTE.

3.5.2 Spaces in Commands

Spaces are significant characters. The following rules apply to their use in lines.

a. There may be a SP immediately preceding eol only if the line ends with a comment. (Since ls may immediately precede eol, this rule does not apply to the SP which may stand for ls.)

b. If a command instance contains at least one argument, the command word or postcond is followed by exactly one space; if the command is not the last of the line, or if a comment follows, the command is followed by exactly one space.

c. If a command instance contains no argument and it is not the last command of the line, or if a comment follows, the command word or postcond is followed by exactly two spaces; if it is the last command of the line and no comment follows, the command word or postcond is immediately followed by eol.

3.5.3 Comments

If a semicolon appears in the command word initial-letter position, it is the start of a comment. The remainder of the line to eol must consist of graphics only, but is otherwise ignored and nonfunctional.

comment ::= ; [graphic] ...

3.5.4 format in READ and WRITE

The format, which can appear in READ and WRITE commands, specifies output format control. The parameters of format are processed one at a time, in left-to-right order.

The parameters, which need not be separated by commas when occurring in a single instance of format, may take the following forms.

! causes a "new line" operation on the current device. Its effect is the equivalent of writing CR LF on a pure ASCII device. In addition, $X is set to 0 and 1 is added to $Y.

causes a "top of form" operation on the current device. Its effect is the equivalent of writing CR FF on a pure ASCII device. In addition, $X and $Y are set to 0. When the current device is a display, the screen is blanked and the cursor is positioned at the upper left-hand corner.

? intexpr produces an effect similar to "tab to column intexpr". If $X is greater than or equal to intexpr, there is no effect. Otherwise, the effect is the same as writing (intexpr - $X) spaces.

3.5.5 Side Effects on $X and $Y

As READ and WRITE transfer characters one at a time, certain characters or character combinations represent device control functions, depending on the identity of the current device. To the extent that the supervisory function can detect these control characters or character sequences, they will alter $X and $Y as follows.

graphic:	add 1 to $X
backspace:	set $X = max($X-1,0)
line feed:	add 1 to $Y
carriage return:	set $X = 0
form feed:	set $Y = 0, $X = 0

3.5.6 Timeout

The OPEN, LOCK, and READ commands employ an optional timeout specification, associated with the testing of an external condition.

timeout ::= : numexpr

If the optional timeout is absent, the command will proceed if the condition, associated with the definition of the command, is satisfied; otherwise, it will wait until the condition is satisfied and then proceed. $T will not be altered if the timeout is absent.

If the optional timeout is present, the value of numexpr must be nonnegative. If it is negative, the value 0 is used. numexpr denotes a t-second timeout, where t is the value of numexpr.

If t = 0, the condition is tested. If it is true, $T is set to 1; otherwise, $T is set to 0. Execution proceeds without delay.

If t is positive, execution is suspended until the condition is true, but in any case no longer than t seconds. If at the time of resumption of execution the condition is true, $T is set to 1; otherwise, $T is set to 0.

3.5.7 Line References

The DO and GOTO commands, as well as the $TEXT function, contain in their arguments means for referring to particular lines within any routine (in the case of DO and GOTO) or within the routine executing the line reference (in the case of $TEXT). This section describes the means for making line references.

Any line in a given routine may be denoted by mention of a label which occurs in a defining occurrence on or prior to the line in question.

lineref ::= dlabel [+ intexpr]

dlabel ::= label | @ expratom V dlabel

If + intexpr is absent, the line denoted by lineref is the one containing label in a defining occurrence. If + intexpr is present and has the value $n \geq 0$, the line denoted is the nth line after the one containing label in a defining occurrence. A negative value of intexpr is erroneous. When label is an instance of intlit, leading zeros are significant to its spelling.

In the context of DO or GOTO, either of the following conditions is erroneous.

a. A value of intexpr so large as not to denote a line within the bounds of the given routine.
b. A spelling of label which does not occur in a defining occurrence in the given routine.

In any context, reference to a particular spelling of label which occurs more than once in a defining occurrence in the given routine will have undefined results.

DO and GOTO can refer to a line in a routine other than that in which they occur; this requires a means of specifying a routine name.

routineref ::= | routinename | @ expratom V routineref |

The total line specification in DO and GOTO is in the form of an entryref.

entryref ::= | lineref [^ routineref] | ^ routineref |

If the delimiter ^ is absent, the routine being executed is implied. If the lineref is absent, the first line is implied.

3.5.8 Command Argument Indirection

Indirection is available for evaluation of either individual command arguments or contiguous sublists of command arguments. The opportunities for indirection are shown in the syntax definitions accompanying the command descriptions.

Typically, where a command word carries an argument list, as in

COMMANDWORD ␣ L argument ,

the argument syntax will be expressed as

argument ::= | individual argument syntax | @ expratom V L argument |

This formulation expresses the following properties of argument indirection.

a. Argument indirection may be used recursively.
b. A single instance of argument indirection may evaluate to one complete argument or to a sublist of complete arguments.

Unless the opposite is explicitly stated, the text of each command specification describes the arguments after all indirection has been evaluated.

3.6 Command Definitions

The specifications of all commands follow.

3.6.1 BREAK

B[REAK] postcond | [␣]
argument syntax unspecified |

BREAK provides an access point within the standard for nonstandard programming aids. BREAK without arguments suspends execution until receipt of a signal, not specified here, from a device.

3.6.2 CLOSE

C[LOSE] postcond ␣ L closeargument

closeargument	::=	expr [: deviceparameters] @ expratom V L closeargument
deviceparameters	::=	expr (expr [: [expr]] ...)

The value of the first expr of each closeargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr is left to the implementor. The deviceparameters may be used to specify termination procedures or other information associated with relinquishing ownership, in accordance with implementor interpretation.

Each designated device is released from ownership. If a device is not owned at the time that it is named in an argument of an executed CLOSE, the command has no effect upon the ownership and the values of the associated parameters of that device. Device parameters in effect at the time of the execution of CLOSE are retained for possible future use in connection with the device to which they apply. If the current device is named in an argument of an executed CLOSE, the implementor may choose to execute implicitly OPEN P USE P, where P designates a predetermined default device. If the implementor chooses otherwise, $IO is given the empty value.

3.6.3 DO

D[O] postcond ␣ L doargument

doargument	::=	entryref postcond
		@ expratom V L doargument

DO is a generalized subroutine call. Each doargument is executed, one at a time in left-to-right order. Execution of a doargument is described below.

a. If postcond is present and false, execution of the doargument is complete at this point. If postcond is absent, or present and true, proceed to the following step.

b. Before proceeding to the next argument of this DO or to the command following this DO, execution continues at the left end of the line specified by the entryref. Execution returns to the argument or command following this argument upon encountering an executed QUIT or eor not within the scope of a subsequently executed doargument or FOR. The scope of this doargument extends to the execution of that QUIT or eor.

3.6.4 ELSE

E[LSE] [␣]

If the value of $T is 1, the remainder of the line to the right of the ELSE is not executed. If the value of $T is 0, execution continues normally at the next command.

3.6.5 FOR

F[OR] ␣ lvn = L forparameter

forparameter ::= | exprl
numexprl : numexpr2 : numexpr3
numexprl : numexpr2 |

The "scope" of this FOR command begins at the next command following this FOR on the same line and ends just prior to the eol on this line.

FOR specifies repeated execution of its scope for different values of the local variable lvn, under successive control of the forparameters, from left to right. Any expressions occurring in lvn, such as might occur in subscripts or indirection, are evaluated once per execution of the FOR, prior to the first execution of any forparameter.

For each forparameter, control of the execution of the scope is specified as follows. (Note that A, B, and C are hidden temporaries.)

a. If the forparameter is of the form exprl.

1. Set lvn = exprl.
2. Execute the scope once.
3. Processing of this forparameter is complete.

b. If the forparameter is of the form
numexprl : numexpr2 : numexpr3
and numexpr2 is nonnegative.

1. Set A = numexprl.
2. Set B = numexpr2.
3. Set C = numexpr3.
4. Set lvn = A.
5. If lvn > C, processing of this forparameter is complete.
6. Execute the scope once.
7. If lvn > C-B, processing of this forparameter is complete.
8. Otherwise, set lvn = lvn + B.
9. Go to 6.

c. If the forparameter is of the form
numexprl : numexpr2 : numexpr3
and numexpr2 is negative.

1. Set A = numexpr1.
2. Set B = numexpr2.
3. Set C = numexpr3.
4. Set lvn = A.
5. If lvn < C, processing of this forparameter is complete.
6. Execute the scope once.
7. If lvn < C-B, processing of this forparameter is complete.
8. Otherwise, set lvn = lvn + B.
9. Go to 6.

d. If the forparameter is of the form
numexpr1 : numexpr2.

1. Set A = numexpr1.
2. Set B = numexpr2.
3. Set lvn = A.
4. Execute the scope once.
5. Set lvn = lvn + B.
6. Go to 4.

Note that form d. specifies an endless loop. Termination of this loop must occur by execution of a QUIT or GOTO within the scope of the FOR. These two termination methods are available within the scope of a FOR independent of the form of forparameter currently in control of the execution of the scope; they are described below. Note also that no forparameter to the right of one of form d. can be executed.

Note that if the scope of a FOR (the "outer" FOR) contains an "inner" FOR, one execution of the scope of the outer FOR encompasses all executions of the scope of the inner FOR corresponding to one complete pass through the inner FOR's forparameter list.

Execution of a QUIT within the scope of a FOR has two effects.

a. It terminates that particular execution of the scope at the QUIT; commands to the right of the QUIT are not executed.

b. It causes any remaining values of the forparameter in control at the time of execution of the QUIT, and the remainder of the forparameters in the same forparameter list, not to be calculated and the scope not to be executed under their control.

In other words, execution of QUIT effects the immediate termination of the innermost FOR whose scope contains the QUIT.

Execution of GOTO effects the immediate termination of all FORs in the line containing the GOTO, and it transfers execution control to the point specified.

3.6.6 GOTO

G[OTO] postcond ␣ L gotoargument

gotoargument ::= doargument

GOTO is a generalized transfer of control. If provision for a return of control is desired, DO may be used.

Each gotoargument is examined, one at a time in left-to-right order, until the first one is found whose postcond is either absent, or present and true. If no such gotoargument is found, control is not transferred and execution continues normally. If such a gotoargument is found, execution continues at the start of the line it specifies.

See 3.6.5 for a discussion of additional effects of GOTO when executed within the scope of FOR.

3.6.7 HALT

H[ALT] postcond [␣]

First, LOCK with no arguments is executed. Then, execution of this process is terminated.

3.6.8 HANG

H[ANG] postcond ␣ L hangargument

hangargument ::= | intexpr |
| @ expratom V L hangargument |

Let t be the value of intexpr. If t $\leq$ 0, HANG has no effect. Otherwise, execution is suspended for t seconds.

3.6.9 IF

I[F] | [␣] |
| ␣ L ifargument |

ifargument ::= | tvexpr |
| @ expratom V L ifargument |

In its argumentless form, IF is the inverse of ELSE. That is, if the value of $T is 0, the remainder of the line to the right of the IF is not executed. If the value of $T is 1, execution continues normally at the next command.

If exactly one argument is present, the value of tvexpr is placed into $T; then the function described above is performed.

IF with n arguments is equivalent in execution to n IFs, each with one argument, with the respective arguments in the same order. This may be thought of as an implied "and" of the conditions expressed by the arguments.

3.6.10 KILL

K[ILL] postcond [␣] | ␣ L killargument

killargument ::= glvn | (L lvn) | @ expratom V L killargument

The three argument forms of KILL are given the following names.

a. Empty argument list: Kill All.
b. glvn: Selective Kill.
c. (L lvn): Exclusive Kill.

Killing the variable M sets $D(M) = 0 and causes the value of M to be undefined. Any attempt to obtain the value of M while it is undefined is erroneous. The value of M remains undefined until M appears to the left of the delimiter = in an executed SET command. $D(M) remains 0 until M or a descendant of M appears to the left of the delimiter = in an executed SET command. Killing a variable whose $D = 0 has no effect.

To kill a variable with the unsubscripted name N also kills all subscripted variables with the same name N.

To kill an m-tuply subscripted variable N(v1, v2, ..., vm) with name N and subscript values v1, v2, ..., vm also kills all n-tuply subscripted variables N(v1, v2, ..., vm, vm+1, ..., vn), for all n > m, with the same N and identical values for the first m subscripts. (These derived n-tuply subscripted variables are called the "descendants" of the m-tuply subscripted variable.)

The Kill All form kills all local variables.

The Selective Kill form kills the variables named.

In the Exclusive Kill form lvn must not contain subscripts, although lvn may have descendants. Exclusive Kill kills all local variables except those named and their descendants.

If M is not killed but N, a descendant of M, is killed, the killing does not effect a change to the value of $D(M).

3.6.11 LOCK

```
LOCK postcond  |  [ ␣ ]                                                    |
               |  ␣ L lockargument                                         |

lockargument ::= |  | nref       |                    [ timeout ] |
                 |  | ( L nref ) |                                |
                 |  @ expratom V L lockargument                    |

nref ::= |  [^] name [( L expr )]  |
         |  @ expratom V nref      |
```

LOCK provides a generalized interlock facility available to concurrently executing MUMPS processes to be used as appropriate to the applications being programmed. Execution of LOCK is not affected by, nor does it directly affect, the state or value of any global or local variable, or the value of the naked indicator.

Each lockargument specifies a subspace of the total MUMPS name space for which the executing process seeks to make an exclusive claim; the details of this subspace specification are given below. Prior to evaluating and executing each lockargument, LOCK first unconditionally removes any prior claim on any portion of the name space made by the process as the result of a prior execution of LOCK. Then, if a lockargument is present, an attempt is made to claim the entire subspace defined by the lockargument. If this subspace does not intersect the union of all other subspaces claimed at this instant by all other processes defined by the implementor as sharing the interlock facility, the claim is successfully established and execution proceeds to the next lockargument or command. If the subspace defined by the lockargument intersects any other claimed subspace, execution of this process is suspended until all interfering claims are removed by one or more other processes, or, when a timeout is present, until the timeout expires, if that occurs first.

The subspace defined by one lockargument is claimed effectively all at once or not at all; thus, the observance of appropriate conventions on the use of the name space by all concurrently executing processes can eliminate the possibility of races and deadlocks.

If a timeout is present, the condition reported by $T upon resumption of execution is the successful establishment of the claim. If no timeout is present, execution of the lockargument does not change $T.

The subspace of the total name space defined by each lockargument is the union of the subspaces defined by each of the name references nref in the lockargument. Each nref specifies its subspace as follows.

a. If the occurrence of nref is unsubscripted, then the subspace is the set of the following points: one point for the unsubscripted variable name nref and one point for each subscripted variable name N(s1, ..., si) for which N has the same spelling as nref.
b. If the occurrence of nref is subscripted, then the subspace is the set of the following points: one point for the spelling of nref after all subscripts have been evaluated and one point for each descendant of nref. (See KILL for a definition of descendant.)

3.6.12 OPEN

O[PEN] postcond ␣ L openargument

```
openargument    ::=  | expr [ : openparameters ]          |
                     | @ expratom V L openargument        |

openparameters  ::=  | deviceparameters [ timeout ]       |
                     | timeout                            |
```

The value of the first expr of each openargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr or of any exprs in deviceparameters is left to the implementor. (See 3.6.2 for the syntax specification of deviceparameters.)

The OPEN command is used to obtain ownership of a device, and does not affect which device is the current device or the value of $IO. (See the discussion of USE in 3.6.16.)

For each openargument, the OPEN command attempts to seize exclusive ownership of the specified device. OPEN performs this function effectively instantaneously as far as other processes are concerned; otherwise, it has no effect regarding the ownership of devices and the values of the device parameters. If a timeout is present, the condition reported by $T is the success of obtaining ownership. If no timeout is present, the value of $T is not changed and process execution is suspended until seizure of ownership has been successfully accomplished.

Ownership is relinquished by execution of the CLOSE command. When ownership is relinquished, all device parameters are retained. Upon establishing ownership of a device, any parameter for which no specification is present in the openparameters is given the value most recently used for that device; if none exists, an implementor-defined default value is used.

3.6.13 QUIT

Q[UIT] postcond [␣]

The end-of-routine mark eor is equivalent to an unconditional QUIT. If the last command of the routine is executed in such a manner as not to transfer control, or if the last command of the routine is an executed DO and control is returned, then the effect of executing off the end of the routine is to execute the QUIT which is implied by the eor.

The effect of executing QUIT in the scope of FOR is fully discussed in 3.6.5. Note that eor never occurs in the scope of FOR.

If an executed QUIT is not in the scope of FOR, then it is in the scope of some doargument or xargument, if not explicitly then implicitly, because the initial activation of a process may be thought of as arising from execution of a DO naming the first executed routine of that process. The effect of executing a QUIT in the scope of a doargument or xargument is to return control to the most recently executed doargument or xargument to which control has not yet been returned by a QUIT. What is executed immediately following the QUIT is the command, doargument, or xargument immediately following the doargument or xargument which most recently transferred control and to which control has not yet been returned. Thus, executed doarguments and xarguments are added to a list of pending returns from which execution of a QUIT (not in the scope of FOR) removes entries in last-in, first-out order.

3.6.14 READ

R[EAD] postcond ␣ L readargument

```
readargument ::= | strlit                          |
                 | format                          |
                 | lvn [timeout]                   |
                 | * lvn [timeout]                 |
                 | @ expratom V L readargument     |
```

The readarguments are executed, one at a time, in left-to-right order.

The top two argument forms cause output operations to the current device; the next two cause input from the current device to the named local variable. If no timeout is present, execution will be suspended until the input message is explicitly terminated. (See 3.6.16 for a definition of "current device".)

If a timeout is present, it is interpreted as a t-second timeout, and execution will be suspended until the input message is explicitly terminated, but in any case no longer than t seconds. If $t \leq 0$, $t = 0$ is used.

When a timeout is present, $T is affected as follows. If the input message has been explicitly terminated at or before the time at which execution resumes, $T is set to 1; otherwise, $T is set to 0.

When the form of the argument is *lvn [timeout], the input message is by definition one character long, and it is explicitly terminated by the entry of one character, which is not necessarily from the ASCII set. The value given to lvn is an integer; the mapping between the set of input characters and the set of integer values given to lvn may be defined by the implementor in a device-dependent manner. If timeout is present and the timeout expires, lvn is given the value -1.

When the form of the argument is lvn [timeout], the input message is a string of arbitrary length which is terminated by an implementor-defined procedure, which may be device-dependent. If timeout is present and the timeout expires, the value given to lvn is the string entered prior to expiration of the timeout; otherwise, the value given to lvn is the entire string.

When the form of the argument is strlit, that literal is output to the current device, provided that it accepts output.

When the form of the argument is format, the output actions defined in 3.5.4 are executed.

$X and $Y are affected by READ the same as if the command were WRITE with the same argument list (except for timeouts) and with each expr value in each writeargument equal, in turn, to the final value of the respective lvn resulting from the READ.

3.6.15 SET

S[ET] postcond ␣ L setargument

```
                                  | glvn          |
setargument  ::=                  | ( L glvn )    | = expr
                                  | @ expratom V L setargument
```

SET is the general means for explicitly assigning values to variables. Each setargument assigns one value, defined by its expr, to each of one or more variables, each named by one glvn.

Each setargument is executed one at a time in left-to-right order. The execution of one setargument occurs in the following order.

a. The glvns to the left of the = are scanned in left-to-right order and all subscripts are evaluated, in left-to-right order within each glvn.

b. The expr to the right of the = is evaluated.

c. The value of expr is given to each glvn, in left-to-right order. For each subscripted glvn of the form N(v1, v2, ..., vn), each variable M whose name is of the form N(v1, v2, ..., vm) for all m < n, as well as the unsubscripted variable N, will be affected as follows.

   1. If M already has a "pointer", that is, if $D(M) has a value of 10 or 11, no change is made to the value of $D(M).

   2. If M has no pointer, that is, if $D(M) has a value of 0 or 1, then it is given a pointer. If $D(M) was 0 it becomes 10, and if $D(M) was 1 it becomes 11.

   The $D value of the glvn itself is changed as follows:

```
 0  becomes  1
 1  remains  1
10  becomes 11
11  remains 11.
```

   That is, the pointer status is not altered, but the variable's value becomes defined. If the glvn is a global variable, the naked indicator is set at the time that the glvn is given its value. If the glvn is a naked reference, the reference to the naked indicator to determine the name and initial subscript sequence occurs just prior to the time that the glvn is given its value.

3.6.16 USE

U[SE] postcond ␣ L useargument

useargument ::= | expr [: deviceparameters] |
| @ expratom V L useargument |

The value of the first expr of each useargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr or of any exprs in deviceparameters is left to the implementor. (See 3.6.2 for the syntax specification of deviceparameters.)

Before a device can be employed in conjunction with an input or output data transfer it must be designated, through execution of a USE command, as the "current device". Before a device can be named in an executed useargument, its ownership must have been established through execution of an OPEN command.

The specified device remains current until such time as a new USE command is executed. As a side effect of employing expr to designate a current device, $IO is given the value of expr.

Specification of device parameters, by means of the exprs in deviceparameters, is normally associated with the process of obtaining ownership; however, it is possible, by execution of a USE command, to change the parameters of a device previously obtained.

Distinct values for $X and $Y are retained for each device. The special variables $X and $Y reflect those values for the current device. When the identity of the current device is changed as a result of the execution of a USE command, the values of $X and $Y are saved, and the values associated with the new current device are then the values of $X and $Y.

3.6.17 VIEW

V[IEW] arguments unspecified

VIEW makes available to the implementor a mechanism for examining machine-dependent information. It is to be understood that routines containing the VIEW command may not be portable.

3.6.18 WRITE

W[RITE] postcond ␣ L writeargument

```
writeargument ::= | format                          |
                  | expr                            |
                  | * intexpr                       |
                  | @ expratom V L writeargument    |
```

The writearguments are executed, one at a time, in left-to-right order. Each form of argument defines an output operation to the current device.

When the form of argument is format, the output actions defined in 3.5.4 are executed. Each character of output, in turn, affects $X and $Y as described in 3.5.4 and 3.5.5.

When the form of argument is expr, the value of expr is sent to the device. The effect of this string at the device is defined by the ASCII standard and conventions. Each character of output, in turn, affects $X and $Y as described in 3.5.5.

When the form of the argument is *intexpr, one character, not necessarily from the ASCII set and whose code is the number represented in decimal by the value of intexpr, is sent to the device. The effect of this character at the device may be defined by the implementor in a device-dependent manner.

3.6.19 XECUTE

X[ECUTE] postcond ␣ L xargument

xargument ::= | expr postcond | @ expratom V L xargument |

XECUTE provides a means of executing MUMPS code which arises from the process of expression evaluation.

Each xargument is executed one at a time in left-to-right order. If the postcond in the xargument is false, the xargument has no effect. Otherwise, if the value of expr is x, execution of the xargument is equivalent to execution of DO y, where y is the spelling of an otherwise unused label attached to the following two-line subroutine considered to be a part of the currently executing routine.

```
y   ls   x    eol
    ls   QUIT eol
```

3.6.20 Z

Z[unspecified] arguments unspecified

All command words in a given implementation which are not defined in the standard are to begin with the letter Z. This convention protects the standard for future enhancement.

MUMPS DEVELOPMENT COMMITTEE

Extensions to the MUMPS Language Standard

A Type A Release of the MUMPS Development Committee

Produced by MDC Subcommittee #1,
Subcommittee on Standard Specifications
and Language Development

The reader is hereby notified that the following language specification has been approved by the MUMPS Development Committee but that it may be a partial specification which relies on information appearing in many parts of the MUMPS specifications. This specification is dynamic in nature, and the changes reflected by this approved change may not correspond with the latest specification available.

Because of the evolutionary nature of MUMPS specifications, the reader is further reminded that changes are likely to occur in the specification released herein prior to a complete republication of MUMPS specifications.

FOREWORD

This document is a collection of extensions to the MUMPS Language Standard which have been discussed and approved by both MDC Subcommittee #1 and the full MUMPS Development Committee.

The ideas embodied in these proposals represent the current consensus of the members of MDC in areas of the MUMPS language where additional capabilities have been deemed greatly beneficial. The purpose of this document is to give present and future implementors guidelines for implementing these extensions.

The format of the presentation is to show the entire affected portion of the MUMPS Language Standard, with the changes made shown by vertical lines in the outside margins (left or right). The section numbers used in this document correspond to those used in the MUMPS Language Standard. Each new extension begins on a new page.

Part I

3.2.8 Functions function

$A[SCII](expr)

produces an integer value as follows:

a. -1 if the value of expr is the empty string.
b. Otherwise, an integer n associated with the leftmost character of the value of expr, such that $A($C(n)) = n.

$C[HAR](L intexpr)

returns a string whose length is the number of argument expressions which have nonnegative values. Each intexpr in the closed interval [0,127] maps into the ASCII character whose code is the value of intexpr; this mapping is order-preserving. Each negative-valued intexpr maps into no character in the value of $C.

Formally approved by SC #1 of the MDC on
12 May 1980 by a 11-0 vote

$D[ATA](glvn)

returns a nonnegative integer which is a characterization of the variable named. The value of the integer is p+d, where:

d = 1 if the named variable has a defined value (that is, it exists);
d = Ø otherwise;
p = 1Ø if either:

a. The named variable contains no subscripts, and there exists a subscripted variable with the same name, or
b. The named variable contains n subscripts, and there exists a subscripted variable with m > n subscripts whose first n subscript values are the same as the values of those in the named variable;

p = Ø otherwise.

Formally approved by SC #1 of the MDC on
12 May 1980 by a 11-Ø vote

$E[XTRACT](expr)

returns the first (leftmost) character of the value of expr. If the value of expr is the empty string, the empty string is returned.

$E[XTRACT](expr1 , intexpr2)

Let s be the value of expr1, and let m be the integer value of intexpr2. $E(s,m) returns the mth character of s. If m is less than 1 or greater than $L(s), the value of $E is the empty string. (1 corresponds to the leftmost character of s; $L(s) corresponds to the rightmost character.)

$E[XTRACT](expr1 , intexpr2 , intexpr3)

Let n be the integer value of intexpr3. $E(s,m,n) returns the string between positions m and n of s. The following cases are defined:

a. m > n. Then the value of $E is the empty string.
b. m = n. $E(s,m,n) = $E(s,m).
c. m < n ≤ $L(s). $E(s,m,n) = $E(s,m) concatenated with $E(s,m+1,n). That is, using the concatenation operator _ of Section 3.3.5, $E(s,m,n) = $E(s,m) _ $E(s,m+1,n).
d. m < n and $L(s) < n. $E(s,m,n) = $E(s,m,$L(s)).

Formally approved by SC #1 of the MDC on
9 November 1978 by a 10-0 vote

$J[USTIFY](exprl , intexpr2)

returns the value of exprl right-justified in a field of intexpr2 spaces. Let m be $L(exprl) and n be the value of intexpr2. The following cases are defined:

a. m ≥ n. Then the value returned is exprl.
b. Otherwise, the value returned is S(n-m) concatenated with exprl, where S(x) is a string of x spaces.

$J[USTIFY](numexprl , intexpr2 , intexpr3)

returns an edited form of the number numexprl. Let r be the value of numexprl after rounding to intexpr3 fraction digits, including possible trailing zeros. (If intexpr3 is the value 0, r contains no decimal point.) The value returned is $J(r, intexpr2). Note that if the value of numexprl is a proper fraction, i.e., between -1 and 1 but not 0, the result of $J does not have a zero to the left of the decimal point (see Section 3.2.5). Negative values of intexpr3 are reserved for future extensions of the $JUSTIFY function.

Formally approved by SC #1 of the MDC on
9 November 1978 by a 9-0 vote

$L[ENGTH](expr)

returns an integer which is the number of characters in the value of expr. If the value of expr is the empty string, $L(expr) returns the value 0.

$L[ENGTH](expr1 , expr2)

returns the number plus one of nonoverlapping occurrences of expr2 in expr1. If the value of expr2 is the empty string, then $L returns the value 0.

Formally approved by SC #1 of the MDC on
12 May 1980 by a 11-0 vote

$N[EXT](glvn)

is included for backward compatibility. The use of the $ORDER function is strongly encouraged in place of $NEXT, as the two functions perform the same operation except for the different starting and ending condition of $NEXT. $N returns a value which is a subscript according to a subscript ordering sequence. This ordering sequence is specified below with the aid of a function, CO, which is used for definitional purposes only, to establish the collating sequence.

CO(s,t) is defined, for strings s and t, as follows:

When t follows s in the ordering sequence, CO(s,t) returns t. Otherwise, CO(s,t) returns s.

Let m and n be strings satisfying the definition of numeric data values (Section 3.2.4.1), and u and v be nonempty strings which do not satisfy this definition. The following cases define the ordering sequence:

a. CO("",s) = s.
b. CO(m,n) = n if and only if n > m.
c. CO(m,u) = u.
d. CO(u,v) = v if and only if v] u.

In words, all strings follow the empty string, numerics collate in numeric order, numerics precede nonnumeric strings, and nonnumeric strings are ordered by the conventional ASCII collating sequence.

Only subscripted forms of lvn and gvn are permitted. Let lvn or gvn be of the form Name(s1, s2, ..., sn) where sn is either a nonnegative integer or -1. Let A be the set of subscripts that follow sn. That is, for all s in A:

a. CO(sn,s) = s and
b. $D(Name(s1, s2, ..., sn-1, s)) is not zero.

Then $N(Name(s1, s2, ..., sn)) returns that value t in A such that CO(t,s) = s for all s not equal to t; that is, all other subscripts which follow sn also follow t.

If no such t exists, -1 is returned.

Note that $N will return ambiguous results for lvn and gvn arrays which have negative numeric subscript values.

Formally approved by SC #1 of the MDC on
6 June 1979 by a 5-2 vote

$O[RDER](glvn)

returns a value which is a subscript according to a subscript ordering sequence. This ordering sequence is specified below with the aid of a function, CO, which is used for definitional purposes only, to establish the collating sequence.

CO(s,t) is defined, for strings s and t, as follows:

When t follows s in the ordering sequence, CO(s,t) returns t. Otherwise, CO(s,t) returns s.

Let m and n be strings satisfying the definition of numeric data values (Section 3.2.4.1), and u and v be nonempty strings which do not satisfy this definition. The following cases define the ordering sequence:

a. CO("",s) = s.
b. CO(m,n) = n if and only if n > m.
c. CO(m,u) = u.
d. CO(u,v) = v if and only if v] u.

In words, all strings follow the empty string, numerics collate in numeric order, numerics precede nonnumeric strings, and nonnumeric strings are ordered by the conventional ASCII collating sequence.

Only subscripted forms of lvn and gvn are permitted. Let lvn or gvn be of the form Name(s1, s2, ..., sn) where sn may be the empty string. Let A be the set of subscripts that follow sn. That is, for all s in A:

a. CO(sn,s) = s and
b. $D(Name(s1, s2, ..., sn-1, s)) is not zero.

Then $O(Name(s1, s2, ..., sn)) returns that value t in A such that CO(t,s) = s for all s not equal to t; that is, all other subscripts which follow sn also follow t.

If no such t exists, $O returns the empty string.

Formally approved by SC #1 of the MDC on
6 June 1979 by a 5-2 vote

$P[IECE](exprl , expr2)

is defined here with the aid of a function, NF, which is used for for definitional purposes only, called "find the position number following the mth occurrence".

NF(s,d,m) is defined, for strings s, d, and integer m, as follows:

When d is the empty string, the result is zero.

When m $\leq$ 0, the result is zero.

When d is not a substring of s, i.e., when $F(s,d) = 0, then the result is $L(s) + $L(d) + 1.

Otherwise, NF(s,d,1) = $F(s,d).

For m > 1, NF(s,d,m) = NF($E(s,$F(s,d),$L(s)),d,m-1) + $F(s,d) - 1.

That is, NF generalizes $F to give the position number of the character to the right of the mth occurrence of the string d in s.

Let s be the value of exprl, and let d be the value of expr2. $P(s,d) returns the substring of s bounded on the right but not including the first (leftmost) occurrence of d.

$P(s,d) = $E(s,0,NF(s,d,1) - $L(d) - 1).

$P[IECE](exprl , expr2 , intexpr3)

Let m be the integer value of intexpr3. $P(s,d,m) returns the substring of s bounded by but not including the m-1th and the mth occurrence of d.

$P(s,d,m) = $E(s,NF(s,d,m-1),NF(s,d,m) - $L(d) - 1).

$P[IECE](exprl , expr2 , intexpr3 , intexpr4)

Let n be the integer value of intexpr4. $P(s,d,m,n) returns the substring of s bounded on the left but not including the m-1th occurrence of d in s, and bounded on the right but not including the nth occurrence of d in s.

$P(s,d,m,n) = $E(s,NF(s,d,m-1),NF(s,d,n) - $L(d) -1).

Note that $P(s,d,m,m) = $P(s,d,m), and that $P(s,d,1) = $P(s,d).

Formally approved by SC #1 of the MDC on
6 June 1979 by a 8-1 vote

```
          |+ intexpr |
$T[EXT]( |           | )
          |  lineref |
```

returns a string whose value is the contents of the line in this routine specified by the argument. Specifically, the entire line, with ls replaced by one SP and eol deleted, is returned.

If the argument of $T is a lineref, the line denoted by the lineref is specified. If the argument is + intexpr, two cases are defined. If the value of intexpr is greater than Ø, the intexprth line of the routine is specified; if the value of intexpr is equal to Ø, the routinename of the routine is specified. An error will occur if the value of intexpr is less than Ø.

If no such line as that specified by the argument exists, an empty string is returned. If the line specification is ambiguous, the results are not defined.

Formally approved by SC #1 of the MDC on
9 November 1978 by a 8-Ø vote

3.6 Command Definitions

3.6.2 CLOSE

C[LOSE] postcond _ L closeargument

```
                                | expr [ : deviceparameters ]   |
      closeargument  ::=        |                               |
                                | @ expratom V L closeargument  |

                                | expr                          |
   deviceparameters  ::=        |                               |
                                | ( [ [ expr ] : ] ... expr )   |
```

The value of the first expr of each closeargument identifies a device (or "file" or "data set"). The interpretation of the value of this expr is left to the implementor. The deviceparameters may be used to specify termination procedures or other information associated with relinquishing ownership, in accordance with implementor interpretation.

Each designated device is released from ownership. If a device is not owned at the time that it is named in an argument of an executed CLOSE, the command has no effect upon the ownership and the values of the associated parameters of that device. Device parameters in effect at the time of the execution of CLOSE are retained for possible future use in connection with the device to which they apply. If the current device is named in an argument of an executed CLOSE, the implementor may choose to execute implicitly the commands OPEN P USE P, where P designates a predetermined default device. If the implementor chooses otherwise, $IO is given the empty value.

Formally approved by SC #1 of the MDC on
7 September 1977 by a 16-0 vote

3.6.8 HANG

H[ANG] postcond _ L hangargument

```
                                | numexpr                          |
hangargument  ::=               |                                  |
                                | @ expratom V L hangargument      |
```

Let t be the value of numexpr. If t $\leq$ Ø, HANG has no effect. Otherwise, execution is suspended for t seconds.

Formally approved by SC #1 of the MDC on
12 May 1980 by a 11-Ø vote

3.6.9.5 JOB

J[OB] postcond _ L jobargument

```
                                | entryref [ : jobparameters ] |
    jobargument  ::=            |                              |
                                | @ expratom V L jobargument   |

                                | expr [ timeout ]                                  |
  jobparameters  ::=            | ( [ [ expr ] : ] ... expr ) [ timeout ]           |
                                | timeout                                           |
```

JOB initiates another MUMPS task at the left end of the line specified by the entryref. The job parameters can be used in an implementation-specific fashion to indicate partition size, principal device, and the like.

Formally approved by SC #1 of the MDC on
7 September 1977 by a 16-Ø vote

Part III

2.2.3 Values of Subscripts

Local variable subscript values are nonempty strings which may only contain characters from the ASCII graphic subset. The length of each subscript is limited to 31 characters. When the subscript value satisfies the definition of a numeric data value (Section 3.2.4.1 of the MUMPS Language Specification), it is further subject to the restrictions of number range given in Section 2.5, and to the set of nonnegative numbers only. The use of subscript values which do not meet these criteria is undefined, except for the use of the empty string as the last subscript of a reference in the context of the $ORDER function, and the use of the value "-1" as the last subscript of a reference in the context of the $NEXT function.

Formally approved by SC #1 of the MDC on
6 June 1979 by a 5-2 vote

8. Other portability requirements

Use of the JOB command is prohibited.

Formally approved by SC #1 of the MDC on
7 September 1977 by a 16-Ø vote

Programmers should exercise caution in the use of noninteger values for the HANG command and in timeouts. In general, the period of actual time which elapses upon the execution of a HANG command or a timeout cannot be expected to be exact. In particular, relying upon noninteger values in these situations can lead to unexpected results.

Formally approved by SC #1 of the MDC on
12 May 1980 by a 11-Ø vote